4-2016

Scalable Parallelization of a Markov Coalescent Genealogy Sampler

Philip E. Davis
Grand Valley State University

Scalable parallelization of a Markov coalescent genealogy sampler

Philip Edmond Davis

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Masters of Science in Computer Information Systems

Padnos College of Engineering and Computing

April 2016

Dedication

This thesis is dedicated to my wife, Denise, who has been a source of support and inspiration. Without you this would not have been possible.

Abstract

Coalescent genealogy samplers are effective tools for the study of population genetics. They are used to estimate the historical parameters of a population based upon the sampling of present-day genetic information. A popular approach employs Markov chain Monte Carlo (MCMC) methods. While effective, these methods are very computationally intensive, often taking weeks to run. Although attempts have been made to leverage parallelism in an effort to reduce runtimes, they have not resulted in scalable solutions. Due to the inherently sequential nature of MCMC methods, their performance has suffered diminishing returns when applied to large-scale computing clusters. In the interests of reduced runtimes and higher quality solutions, a more sophisticated form of parallelism is required. This paper describes a novel way to apply a recently discovered generalization of MCMC for this purpose. The new approach exploits the multiple-proposal mechanism of the generalized method to enable the desired scalable parallelism while maintaining the accuracy of the original technique.

Contents

List of Figures

1 Introduction

One useful application of computers is in solving otherwise intractable mathematical problems. Computers are able to perform computations at an ever-increasing rate of performance. Many tasks that would take an impractical amount of time if performed by a human being can be completed in very short amount of time by a computer. This quantitative difference in capabilities has led to qualitative differences in methods. One area of study in which these new qualitative methods have proven useful is in the field of population genetics. Applying computational analysis to genetic sequence data has produced new information about current and past populations. A particularly useful tool in computational population genetics is the coalescent genealogy sampler. This tool estimates parameters of a population based on a survey of likely genealogical histories.

The coalescent genealogy sampler is an application of a broader class of methods called Markov Chain Monte Carlo (MCMC). MCMC algorithms use clever applications of statistics to select samples from complex probability distributions such as those defined by the coalescent genealogy sampler. Although the intuitions behind MCMC are reasonably straightforward, the statistical analysis of both the process and the results is non-trivial. The nature of MCMC algorithms is such that they are typically employed when direct computation of some result would require an impossible amount of computation time. Although MCMC methods typically require vastly less time to produce an approximation of the result that would be produced by direct computation, they are typically still quite computationally expensive. Coalescent genealogy samplers are no exception to this, requiring a very large amount of computation even for relatively small problems.

Due to the computational intensity of coalescent genealogy samplers, it is desirable that they be parallelized. Modern large-scale computing clusters are capable of bringing a large degree of parallelism to bear. This potentially reduces the runtime of these programs by multiple orders of magnitude. However, current approaches that apply parallelism to coalescent genealogy samplers suffer from a lack of scalability due to inherent features of MCMC methods. The flavor of MCMC typically employed, the standard Metropolis-Hastings algorithm, suffers from a significant sequential initialization that cannot be parallelized. Therefore, achieving

highly-scalable parallelism requires more sophisticated MCMC methods.

This thesis intends to demonstrate that scalable parallelism of coalescent genealogy samplers is possible through a version of the newly-discovered Generalized Metropolis-Hastings algorithm. This algorithm allows the previously sequential initialization stage to be parallelized, removing a significant barrier to efficiently achieving a high degree of parallelism overall and reducing the run-time of these important tools.

2 Background

Coalescent genealogy samplers apply stochastic statistical methods to answer questions in population genetics. In this section, these topics will be briefly covered.

2.1 Statistical Modelling

Many problems in physical science, statistics, and machine learning involve answering questions about systems such that the state of the system at a given time cannot be singularly determined, but rather is stated as a probability distribution across a set of possible states. This set of possible states is the state space. This may be necessary when the state of the system varies probabilistically over time, is part of a large ensemble of states being examined, or when there is simply insufficient information to fully determine the state of the system.

Often, answering questions about these probabilistic systems involves finding the expected value of some feature of the system across the entire state space. This means calculating the mean value of that feature for every state, while weighting by each state's relative probability. This calculation takes the form:

$$E_X[f] = \int_X f(\mathbf{x})p(\mathbf{x})d\mathbf{x}, \tag{1}$$

where X is the state space, $\mathbf{x} \in X$ is some possible state of the system, $f(\mathbf{x})$ is the value of the feature of interest for the state $\mathbf{x}$, $p(\mathbf{x})$ is the distribution of the states and $E_X[f]$ is the expected value of f over all the states of X.

Note that each parameter of the problem being formalized translates to a dimension of the state space. Because these problems typically have a large number of parameters, this means the state space must frequently have a high degree of dimensionality in order to fully describe the possible states of the system. Problems of this form are often too computationally intense to directly calculate. For example, consider the motivational question of determining the probability that a given game of solitaire is winnable.[5] Take X to be the set of all possible initial draws and $\mathbf{x} \in X$ to be a particular draw. Once the deck has been shuffled and cards drawn, the game is either winnable or it is not. Note that the player may still lose a winnable game

through lack of information, but the game was still winnable when started. So whether the game is winnable or not is a function of the outcome of the initial draw. Define a new function f_S as

$$f_S(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} \text{ is winnable} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

The probability of drawing any given game of solitaire is the same as that of drawing any other, so the distribution of states is uniform and is given as

$$p_S(\mathbf{x}) = \frac{1}{|X|} \tag{3}$$

where $|X|$ is the size of the state space, in this case the number of unique games of solitaire. Because the states being described (the drawn games of solitaire) are discrete, Equation (1) reduces to the summation [3, p. 15]

$$E_X[f_S] = \sum_X f_S(\mathbf{x}) p_S(\mathbf{x}) \tag{4}$$

The trivial approach to finding the value of $E[f]$ — the probability that a random game of solitaire is winnable — is to iterate all the possible games of solitaire, determining if each one is winnable. For each one that is winnable, increment a register. Once the task of checking every single possible game of solitaire has been completed, then the register representing the tally of winnable games is divided by the total number of games of solitaire to determine the winnable fraction. However, there are $52! \approx 10^{68}$ different ways that a standard deck of cards can be arranged. Even accounting for symmetries from the interchangeability of different suits, and further reductions due to rules variations, the number of different possible games of solitaire would mean that the trivial approach described would require an impractical amount of computation to arrive at an answer.

Fortunately, it is not necessary to iterate every state. It is possible to calculate a reasonable approximation using a randomly selected sampling of states. This is the intuition behind *Monte Carlo* methods.

2.2 Monte Carlo

In order to address the impracticality of directly calculating results in cases such as the solitaire example above, the Monte Carlo method was developed. Monte Carlo methods use a subset of the state space to generate an approximate result, rather than calculating an exact result for the *entire* state space. The method is to sample this subset of states from the state space using a process that produces the same state distribution as the state space, which is the *target distribution*. For a certain sampling process to produce the same state distribution as the state space means that, for any given state, the frequency with which the process samples that state will be proportional to the state's probability density in the distribution of the state space. For example, if the system is twice as likely to be in state i as state j, then the process will eventually sample i twice as often as state j. To say that this will happen *eventually* means that this is the expected behavior as the number of samples gets very large. In this same way, the frequency of heads in a fair coin flip will *eventually* be 0.5 as the number of coin flip events becomes large.

Suppose N samples are taken to generate the set $\left\{\mathbf{x}^i\right\}_{i=1}^{N}$ where $\mathbf{x}$ is sampled from $p(\mathbf{x})$. For $\mathbf{x}$ to be sampled from $p(\mathbf{x})$ means that for a given $\mathbf{x}$, the probability that $\mathbf{x}$ is selected is equal to $p(\mathbf{x})$. The law of large numbers states that the mean of a set of samples will approach the mean of the population being sampled as the sample grows. Because the probability of a given sampled state is implicitly accounted for in its probability of having been sampled, the expectation of f, the feature of interest, should not be weighted on that probability. So, using already defined terms, the convergence of the approximation can be expressed as

$$\frac{1}{N}\sum_{i}^{N} f(\mathbf{x}_i) \xrightarrow{N\to\infty} \int_X f(\mathbf{x})p(\mathbf{x})d\mathbf{x} = E_X[f]. \tag{5}$$

This shows that larger values of N — larger sample sets —- should produce more accurate results. In other words, sampling more states will produce a better approximation of the expectation. The quality of the approximation can be quantified: the error of the approximation ($|E_X[f] - \frac{1}{N}\sum_i^N f(\mathbf{x}_i)|$) converges to zero at a rate that is inversely proportional to $\sqrt{N}$. [2]

Monte Carlo methods provide a way to estimate the expected value of f in less time than

finding a weighted average of f for the entire state space. In the solitaire example, it is possible for a human being to provide a useful estimate by running through hundreds or even thousands of games of solitaire in a practical time period. So it can be seen that, even in this relatively simple example, an application of Monte Carlo methods reduces the amount of computation required to calculate an answer from an amount that is not feasible for a computer to complete to an amount that is potentially feasible for a human being to complete.

A crucial requirement of applying Monte Carlo methods is to have a process that can sample states with the same probability distribution as the target distribution. In the solitaire example, the probability of any given game being drawn is the same. This makes it fairly easy to sample from the target distribution $p_S(\mathbf{x})$. The cards just need to be arranged at random, since any configuration of cards is just as likely as any other. In fact, a process to sample from this distribution is inherent in the game of solitaire: an ideal shuffle of the deck should produce a random game. This is an example of a uniform distribution, from which it is trivial to sample correctly. There are other distributions for which algorithms exist for easy sampling. For example, gamma distributions, normal distributions and binomial distributions — among others — all lend themselves to easy sampling. However, many problems that might benefit from Monte Carlo methods have target distributions that are much more difficult to sample.

Consider the motivating example described by Metropolis in [20]. In this paper, a substance has been modelled as a square containing N particles which are kept at a constant, uniform temperature. For simplicity's sake, the square is periodic, meaning that a particle that leaves the square immediately enters the square again on the opposite side. The state of the system is described by the configuration of the N particles, which consists of the position and momentum of each. So in the two-dimensional example, each state $\mathbf{x}$ can be described as a $2N + 2N$-dimensional vector and the state space X is $\mathbb{R}^{4N}$. Note that, in contrast to the solitaire example, the states in the particle example are continuous.

In this example, the value of interest is a function f of the position and momentum of the N particles in the system, and so for a given state $\mathbf{x}$ there is a unique $f(\mathbf{x})$. The probability of the particles being in any specific configuration given their temperature is given by the laws of statistical mechanics, and can be determined by the canonical ensemble equation [11] as

follows:

$$p_M(\mathbf{x}) = \frac{1}{Z} e^{-\frac{E}{kt}}. \tag{6}$$

The terms of this function require some explanation. The constant k is Boltzmann's constant and t is the temperature of the system. In this context, E is a function of $\mathbf{x}$ that gives the potential energy of the configuration. The total potential energy of a given configuration is the summation of the pairwise potential energy of the particles. Finally, Z is a normalization constant, calculated by

$$Z = \int_X e^{-\frac{E}{kt}} d\mathbf{x}. \tag{7}$$

The reason for the normalizing constant is to ensure that the probabilities of all possible states of the system sum to one. That this is accomplished can be demonstrated by taking the marginal integration of $p_M(\mathbf{x})$:

$$\int_X p_M(\mathbf{x}) = \int_X \frac{1}{Z} e^{-\frac{E}{kt}} d\mathbf{x} = \frac{1}{Z} \int_X e^{-\frac{E}{kt}} d\mathbf{x} = \frac{Z}{Z} = 1. \tag{8}$$

This formulation reflects the fact that configurations with a higher potential energy are less probable than configurations with a lower potential energy. The means that the particles are more likely to assume evenly distributed configurations than configurations in which they bunch together, with a more marked drop-off in the probabilities of these more highly-entropic states at low temperatures. It is also evident that the solution to Equation 1 for this example requires an integration that is unlikely to be directly computable. Unfortunately, it is also evident that sampling from the target distribution p_M given in Equation 6 using standard methods is not practical, as E (the potential energy of the given configuration) is a function of $2N$ independent variables, i.e. the positions in each dimension of every particle, which expands to a summation of $N^2 + N$ terms. This suggests a more sophisticated sampling method is required in order to apply Monte Carlo Methods to this example. The Markov Chain Monte Carlo (MCMC) method proposed in [20] provides this method of sampling from a target distribution.

2.3 Markov Chain Monte Carlo

The *Markov Chain Monte Carlo* (MCMC) method is able to sample from much more complex distributions than those previously described. It is able to do this by defining a particular kind of process for sampling states out of the state space. This process is called a *Markov chain*.

A Markov chain is fundamentally a process that probabilistically traverses a state space. This means that the chain will visit one state after another, at each state choosing the following, or *successor*, state from a probability distribution across the state space. The defining characteristic of a *Markov* chain is that the probability distribution out of which the successor state is chosen is determined by the current state of the chain alone. In other words, variables such as the history of the chain, the total number of states traversed, and so on have no bearing upon the chance of moving from the current state to any possible successor state. So for a given Markov chain, the probability of moving from a current state i to a different state j — the *transition probability* — is *invariant* over time.

A Markov chain can be used as a sampler by treating the series of states the chain visits as a series of samples. Over time, this series will converge to a particular distribution, which is the *stationary distribution* of the Markov chain. Additionally, if a graph representing the transition probabilities of the Markov chain is both aperiodic and irreducible, then the Markov chain has the property of *ergodicity*.[3, p. 104] In order for a Markov chain to be aperiodic there can be no restrictions on the number of transitions it takes to return to a given state beyond some number of transitions. Irreducible means that it is always possible to go between any two states via some series of transitions. The Markov chain being ergodic means that the chain will always have the same stationary distribution, regardless of the starting state. So an ergodic Markov chain being used as a sampler will always converge to the same distribution. In other words, the Markov chain will eventually be sampling out of the same probability distribution, no matter where the chain starts in the state space.

Figure 1 illustrates a simple Markov chain. Each state A, B, and C, shows the possible transitions that can be made away from the state, as well as the probability of each transition occurring. Notice that the probabilities of all the possible transitions at any given state add up

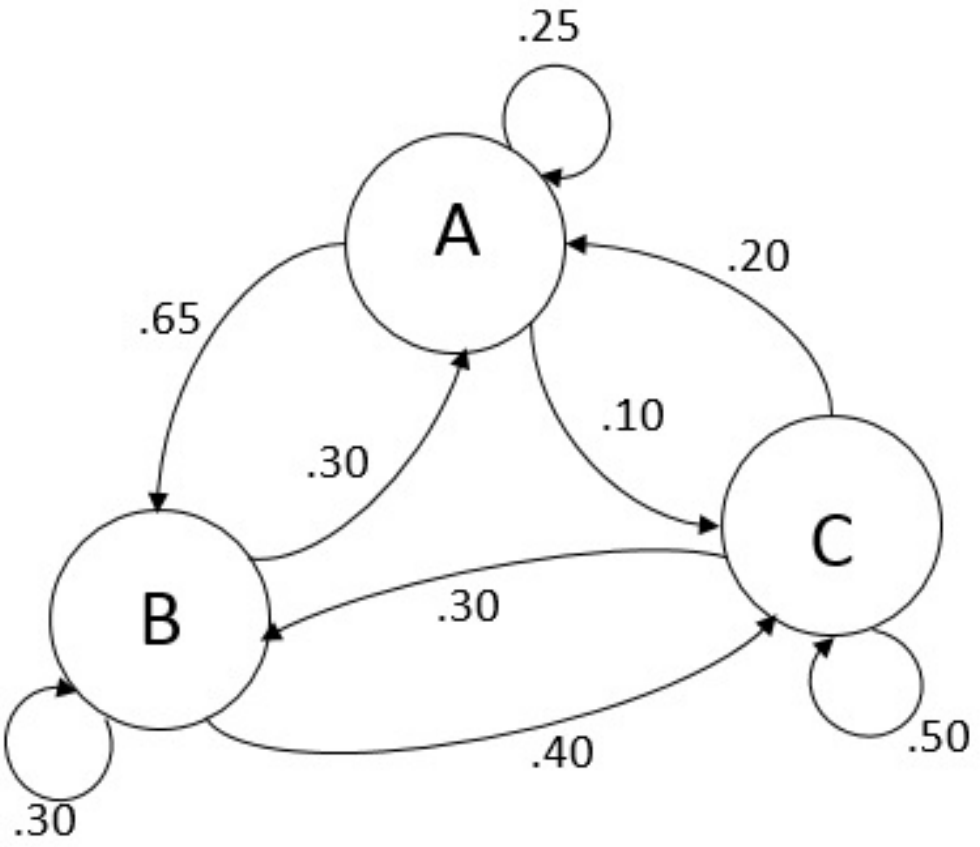

Figure 1: Markov chain state diagram

to one. This Markov chain is trivially aperiodic since any state can transition to itself and is irreducible as there is a series of transitions to get from any state to any other state. Because this Markov chain is both aperiodic and irreducible it is ergodic, which means that there is a unique stationary distribution. The distribution of the states visited by the Markov chain will converge to this distribution over time, regardless of the starting state of the chain.

For example, suppose that the following facts describe the probabilities of sunny, rainy, and cloudy days:

- If it is sunny today, then there is a 50% chance that it will be sunny tomorrow, a 15% chance that it will be rainy tomorrow, and a 35% chance that it will be cloudy tomorrow.
- If it is rainy today, then there is a 10% chance that it will be sunny tomorrow, a 30% chance that is will be rainy tomorrow, and a 60% chance that it will be cloudy tomor-

row.

- If it is cloudy today, then there is a 20% chance that it will be sunny tomorrow, a 25% chance that it will be rainy tomorrow, and a 55% chance that it will be cloudy tomorrow.

These facts describe a process for probabilistically traversing three states: sunny, cloudy, and rainy. Because the distribution of tomorrow's weather is dependent only upon the current state of the weather, this process is a Markov chain. The states are fully connected to each other (and each state to itself), so that this Markov chain is trivially aperiodic and irreducible. There is then a stationary distribution that is determined by the transition probabilities of the Markov chain. Regardless of the weather at the start of the Markov chain, after six days the probability of it being a sunny, rainy, and cloudy day has converged to approximately 25.1%, 23.6%, and 51.1%, respectively. The Markov chain will visit these weather states with the given distribution indefinitely.

MCMC is a particular way of applying Monte Carlo methods that leverages Markov Chains. Using MCMC involves defining a Markov chain with transition probabilities in such a way that the stationary distribution of the Markov chain is the same as the target distribution that must be sampled as part of applying the Monte Carlo method. In the case of the particle example from [20], the target distribution is Equation 6, so the goal is to define a Markov chain with this stationary distribution. In fact, the *Metropolis-Hastings algorithm*[13] sets out a way to define a Markov chain that, given a target distribution, has a stationary distribution that matches that target. In order to elucidate this algorithm, the concept of the state transition probabilities and the target distribution must be formalized. Suppose there are a finite number S of states in the state space $X = \{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_S}\}$. Although this may seem limiting, it is not necessarily so as any parameter of the state must be modelled discretely for computing. For example, a parameter that takes a floating point value will be stored in computer memory and will have a minimum granularity and will only be able to represent a finite number of values. For the purposes of constructing the Metropolis-Hastings algorithm, the granularity may be arbitrarily small. In other words, there are no implementation consequences to viewing the algorithm in this manner. A Markov chain on X (meaning that the states that the Markov chain

can transition to are all in X) can have its state transition probabilities fully defined as a matrix of the form $\mathbf{P} = \{p_{ij}\}$, where p_{ij} is the probability that the Markov chain will transition to state $\mathbf{x}_j$ when the current state is $\mathbf{x}_i$. Note that because a Markov chain must transition to one and only one state at every given step, for all i

$$\sum_{j=1}^{S} p_{ij} = 1. \tag{9}$$

Take the target distribution of the Monte Carlo problem to be $\pi = \{\pi_i\}$ so that π has the same distribution as state space. In the solitaire example, this means that

$$\pi_{S_i} = p_S(\mathbf{x_i}) = \frac{1}{|X|}, \tag{10}$$

while in the particle example, the stationary distribution is defined by

$$\pi_{M_i} = p_M(\mathbf{x_i}) = \frac{1}{Z} e^{-\frac{E}{kt}}. \tag{11}$$

The constraint at the heart of the Metropolis-Hastings algorithm is that the state transition probability matrix $\mathbf{P}$ is defined in such a way that the following constraint, known as reversibility, holds for all i and j:

$$\pi_i p_{ij} = \pi_j p_{ji}. \tag{12}$$

This constraint is significant because Equation 9 combined with Equation 12 produce, for all j,

$$\sum_{i=1}^{S} \pi_i p_{ij} = \sum_{i=1}^{S} \pi_j p_{ji} = \pi_j \sum_{i=1}^{S} p_{ji} = \pi_j, \tag{13}$$

which implies by the principles of matrix multiplication that $\pi\mathbf{P} = \pi$, demonstrating that π is the stationary distribution of the Markov chain defined by $\mathbf{P}$[3, p. 104]. The intuition behind this demonstration follows Metropolis[20]. Consider a large ensemble' of Markov chains all defined by $\mathbf{P}$ and where Equation 12 holds. Then n_i of these chains are in state $\mathbf{x_i}$ and n_j of these chains are in state $\mathbf{x_j}$ so that on the next transition $n_i p_{ij}$ chains will transition from state $\mathbf{x_i}$ to state $\mathbf{x_j}$ and $n_j p_{ji}$ chains will transition from state $\mathbf{x_j}$ to state $\mathbf{x_i}$. Therefore the net flow be-

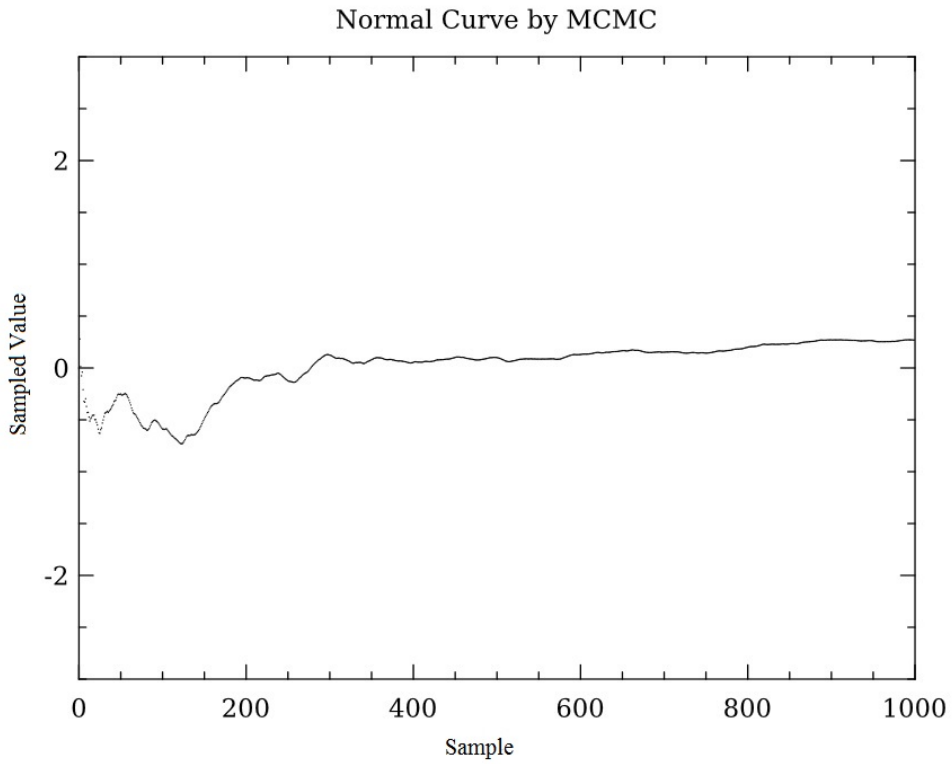

Figure 2: Markov chain burn-in

tween $\mathbf{x_i}$ and $\mathbf{x_j}$ is $n_i p_{ij} n_j p_{ji}$. This net flow will reduce the imbalance between the states, and so the net flow will reduce on further subsequent transitions. When $\frac{n_i}{n_j} = \frac{\pi_i}{\pi_j}$, there will be no net flow and the distribution will be at equilibrium (converged.) Due to the ergodicity of the Markov chain, not only is this equilibrium unique and therefore the stationary distribution, but the distribution of a large ensemble of Markov chains will have the same stationary distribution as a single chain sampling over a large number of transitions[20].

The Metropolis-Hastings algorithm sets out a general method for achieving this using two stages: a proposal mechanism for suggesting transitions in a way that guarantees ergodicity, and an acceptance function that determines how probable it is that a given proposal will be accepted (the chain will sample the proposal.) If the proposal is not accepted, then the next state is the current state, and the current state will be sampled again. By relating the acceptance probability to the ratio of the probability of the current state to that of the proposed state, the stationary distribution of the Markov chain can be shown to be proportional to that of the sample space.[13]

The question of how to initialize a Markov chain is an important one. In many situations,

a randomly selected state will have a very low probability of being sampled from the stationary distribution. In this case, the chain will initially traverse a number of states at a proportion that would bias the estimate produced by the chain before converging to the stationary distribution. Figure 2 shows an example of a Markov chain converging to a normal stationary distribution. It can be seen that the first few hundred samples are quite different from the behavior of the chain upon convergence, and would generate a bias if they were included in the samples. To deal with this, a *burn-in* period is observed in the course of sampling, in which the first several states are not considered viable random samples. The determination of the appropriate length of the burn-in period is difficult to ascertain, and typically the method used to determine when the burn-in period is over will be part of sampler implementation. The trivial solution is to specify a fixed number of samples in the burn-in period, but this risks either providing a biased result if the burn-in time is too short or wasting computation time if the burn-in time is too long. Methods also exist to evaluate if the burn-in period is over while the chain is in progress. One such method is to use a sample statistic such as covariance of the samples to determine if the chain has stabilized, but this can lead to premature termination. A possible counter to the risk of premature termination is to compare the output of multiple chains: if multiple chains are run and produce significantly different results, this indicate insufficient burn-in time. However, running multiple chains is inefficient. The ideal solution to burn-in is to have a random sample of the target distribution available as an initial state. One way to accomplish this is to initiate new chains using the final state of previous chains with the same stationary distribution. The danger in using this procedure is that, improperly used, reusing old chain results can introduce bias[23]. This method also assumes that results from an identical distribution are available, which they may not be. Another way to provide a random sample for initialization is through perfect sampling [10], which is able to produce a single random sample from the target distribution at extremely high computational cost. The downside to perfect sampling is that the high computational cost of generating the perfect sample can exceed the cost of a reasonable burn-in time in many applications. The problem of burn-in is not solved in the general case, and the method of dealing with it is both domain and implementation specific.

2.4 Population Genetics

Population genetics uses genetic information about a population (or a subset of a population) in order to infer hypotheses about the past or present features of that population. By observing the diversity of genetic samples, biologists can make reasonable estimates of parameters of interest such as effective breeding population size, growth or decline rate, migration rate with a different population, and more. By formalizing and characterizing the information content of genetic samples, techniques in machine learning can be applied to infer values for these parameters.

Genetic information is organized into *genes*, where each gene is the information encoding a particular protein. Different individuals can have different variants of this information. Sometimes this results in functionally different versions of the same gene; for example, the gene for hemoglobin antigen has an A form, a B form, and an O form. Additionally, there are variants of each of these forms that do not affect function, but do carry detectably different genetic information. These latter genes are *non-functional variants*. More generally, the different variants of the same gene are called *alleles*. Note that non-functional variants of an allele can only be revealed by discovering differences in DNA sequencing of the different alleles. Additionally, when analyzing allele prevalence in a diploid population, each individual carries two alleles of each gene (which may be the same.)

One way of characterizing a genetic population is by its allele proportion over time. As an example, consider a gene that has only non-functional variants in a population. A gene having only non-functional variants has two implications. First, all members of the population will have the same function from the gene regardless of which allele is present and will therefore have the same survival benefit from that allele. Second, members of the population can be classified by which allele they carry. Furthermore, because genetic information is passed from parent to child in successive generations, there is a dependency between the distributions of alleles from one generation to the next. However, there are elements of uncertainty in reproduction so this dependency is stochastic, rather than absolute. Some portion of any given generation will not reproduce and will not pass on genetic information. Some members will reproduce at a rate disproportionate to other members.

The effect of this uncertainty is the phenomenon known as *genetic drift*. Over time, the proportion of a given allele can increase or decrease based solely on chance accidents of reproduction. Because the distribution of alleles is not independent from generation to generation, a shift in the proportion of an allele will be propagated to future generations. In the extreme case, no members of the population with a particular allele will pass it on to the next generation and so that allele will not be present in future generations. If only one allele remains in the population for a particular gene, then the population is *fixed* at that allele.

Genetic drift accounts for the spread of *non-selective mutations*. A non-selective mutation is one that does not change the survival value of the gene, but still results in a new allele. Initially the allele that results from a mutation will only be present in a single individual in the population. The allele will then then be passed on to progeny in the following generation. If the descendants who have the mutated allele happen to be disproportionately successful in reproducing, then the allele will be present in a more significant portion of the population. The probability of non-selective mutations spreading in a population is a function of the effective size of the population, so inferences about the population can be made by modelling the stochastic processes of genetic drift.

Under certain assumptions, genetic drift can be approximated by the Wright-Fischer model.[12, p. 102] These assumptions include mating in discrete generations and that, for a given member of a generation, there is a uniform distribution across the previous generation in picking its parents. In other words, the probability that a given organism is the parent of a given organism in the next generation is uniform across each organism in both generations, independent of any other descendent relationships. An organism in the earlier generation then has a vanishingly small (but non-zero) chance of being the parent of every organism in the following generation.

The Wright-Fisher model is typically stated for diploid populations, where each organism has two alleles for any given gene. A child organism receives one allele from each parent. So in a population with N organisms, there are $2N$ total allele copies. If a fraction p of the alleles in a generation is allele A, then the probability of there being k copies of allele A in the following generation can be found by observing that the fraction of alleles that aren't A will be

$1 - p$ (call this fraction q.) If, for a given organism in the new generation the probability of parentage is uniform across the previous generation, then the probability of receiving allele A from the previous generation is p, and the probability of receiving a different allele is q. So in a total population of $2N$ alleles, the probability of any particular outcome that has k copies of allele A is

$$p^k q^{(2N-k)}. \tag{14}$$

The number of different possible outcomes with k copies of allele A is a combinatorial problem: pick k from $2N$ where order does not matter. This is

$$\frac{2N!}{k!(2N-k)!}. \tag{15}$$

So the probability of there being k copies of allele A among N offspring is

$$\frac{2N!}{k!(2N-k)!} \times p^k q^(2N-k). \tag{16}$$

When taken as a distribution across the possible values of k, which are from 0 (no copies of allele A are inherited) to $2N$ (no allele other than A is inherited), Equation 16 takes the form of a binomial distribution.

This model of genetic drift can be used to make estimations about the genealogical history of a population. A genealogy can be represented as a tree structure with the various current alleles at the leaves of the tree. Looking backwards in time, the genealogy is a series of coalescent events, in which two alleles or — from a diachronic perspective — *lineages*, descend from a single ancestral lineage. These coalescent events continue backwards until reaching a lineage that is the common ancestor of all lineages. This is the root of the tree structure. The Wright-Fischer model can be used to estimate how frequently these coalescent events occur. There are several factors that influence the probability of a coalescence in a given period. One factor is the number of lineages present during the interval. More lineages increase both the number of *possible* coalescent events and the probability that a coalescent has recently occurred. Additionally, a higher mutation rate μ implies more rapid *divergences*

of lineages and similarly implies that it is more likely that the last *convergence* occurred more recently than a lower mutation rate would. Similarly, a larger breeding population N_e creates more opportunity for mutation, increasing the divergence and, hence, convergence rates. The parameter $\theta = \mu N_e$ is important to population genetics since mathematical models of drift cannot provide information on μ or N_e alone, but can only make estimates about their product, θ.[18]

Assuming the Wright-Fischer model, a probability distribution $p(t)$ can be constructed,[15] which is the probability that the most recent coalescence of k lineages occurred t generations ago:

$$p_k(t) = \left(\frac{2}{\theta}\right) e^{\frac{-k(k-1)t}{\theta}}. \tag{17}$$

Additionally, the posterior probability of a genealogy G given a θ value can be derived from Equation 17 by viewing the genealogy as a sequence of coalescent events backwards in time. Suppose there are n lineages at the present and that t_i is the length of time leading up to the "$i+1$"th coalescent event. So, for example, t_0 would be the length of time between the present and the first coalescent event in the past; t_3 would be the length time between the third and fourth coalescent event, counting backwards in time. Note that for the duration of that time interval, there are $n-i$ lineages. This concept is illustrated in Figure 3. The posterior probability of G is the probability that *all* of its intervals occur, and each coalescent interval length is independent of the other. This means that the posterior can be taken to be the product of the probabilities of each individual interval, which can each be calculated to be[18]:

$$P(G|\theta) = \prod_{i=0}^{n-2} p_{n-i}(t_i) = \prod_{i=0}^{n-2} \left(\frac{2}{\theta}\right) e^{\frac{-(n-i)((n-i)-1)t_i}{\theta}} = \left(\frac{2}{\theta}\right)^{n-1} e^{\left(\sum_{i=2}^{n} \frac{-i(i-1)t_{n-i}}{\theta}\right)}. \tag{18}$$

It is also possible to use statistical models of evolution to gain information from the actual base pair data in sequenced genetic samples in addition to what can be determined from the diversity of alleles. An algorithm is described in [7] for determining the overall likelihood of a genealogical tree G having produced the observed data D. Consider the genealogical tree given in Figure 4 as a possible G. The genealogical data, D, for some base pair are given at the leafs of the tree. Consider the probability distribution $L_{n_i}(X)$ which is the likelihood,

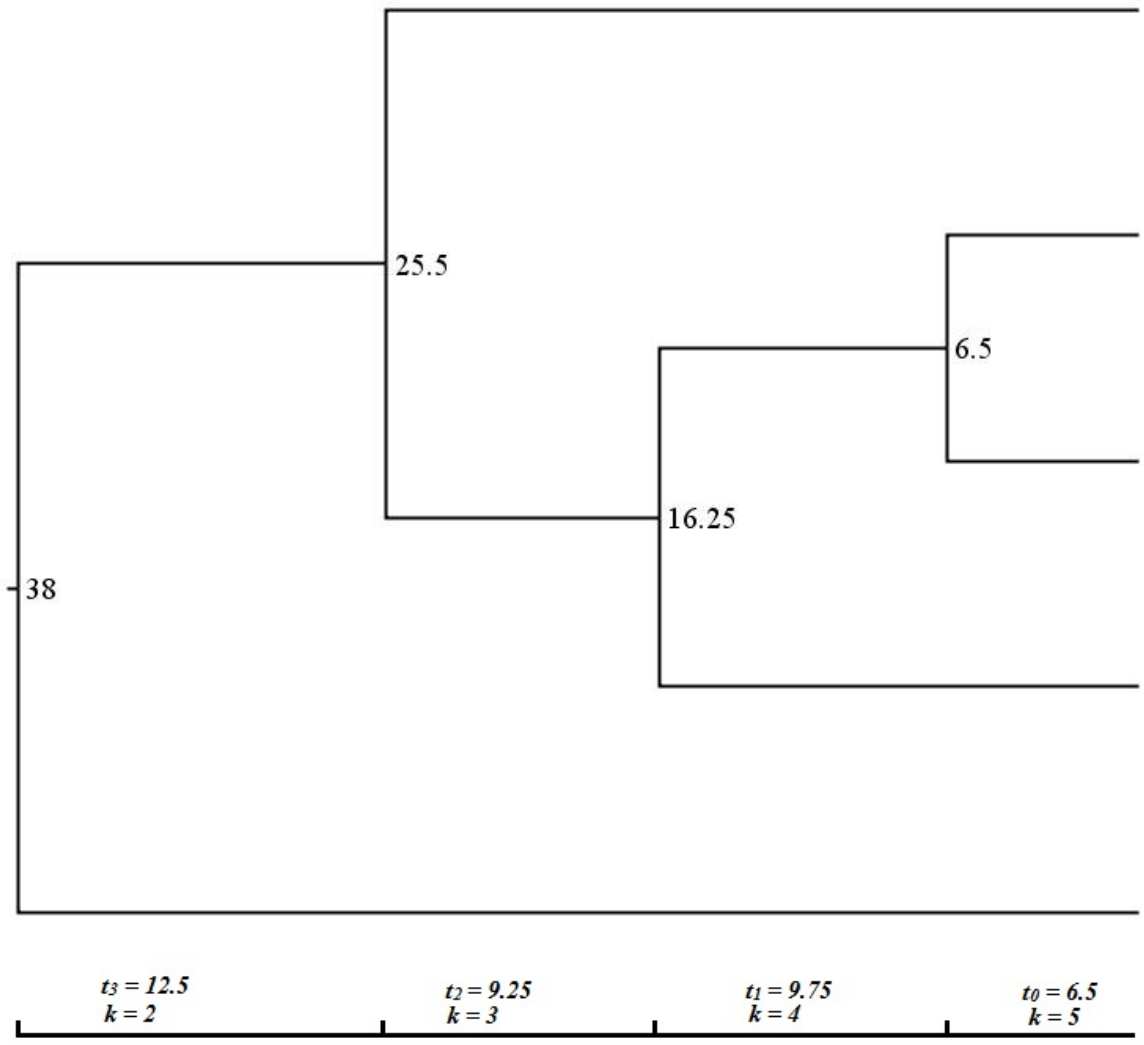

Figure 3: A genealogical tree with intervals labeled. t_i is the length of the interval, and k is the number of lineages through the duration of the interval.

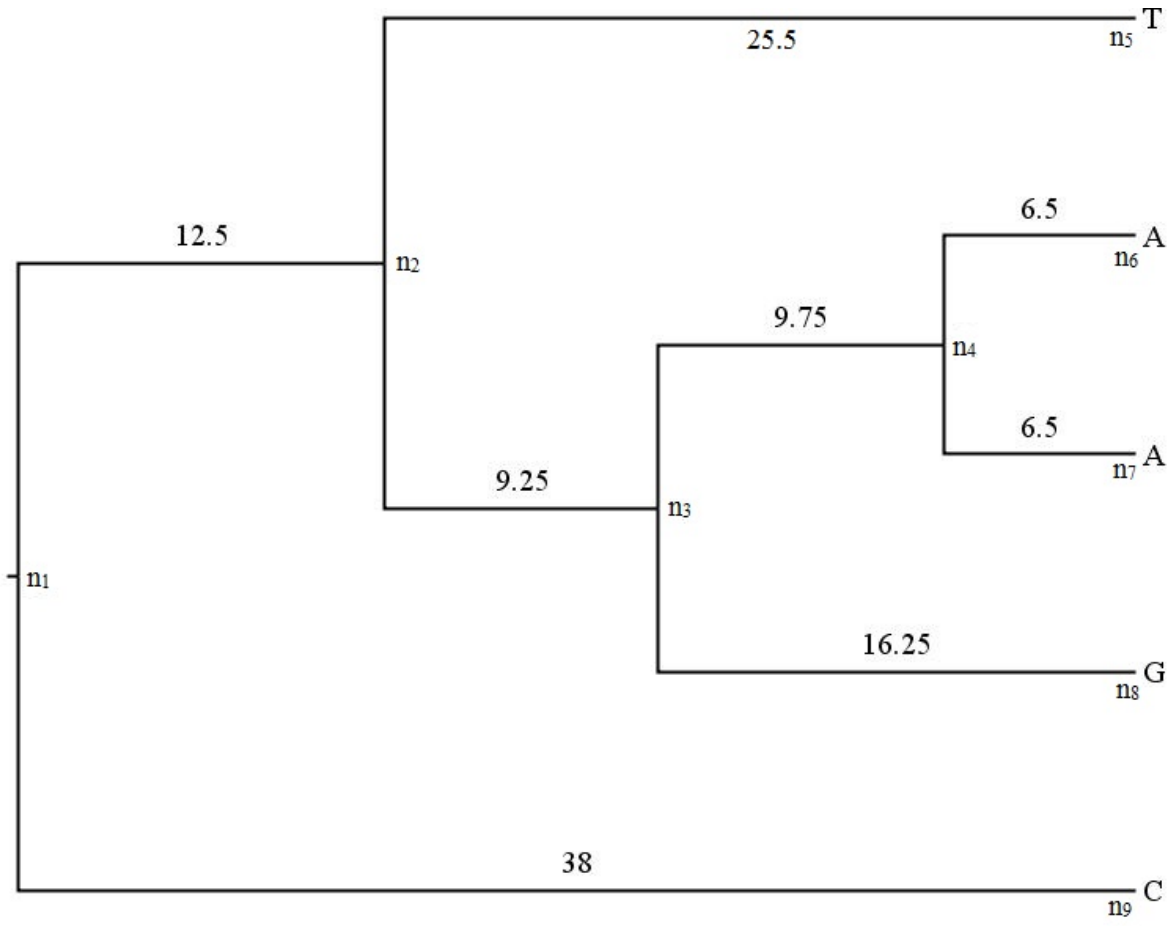

Figure 4: A genealogical tree with sequence data and nodes labeled

given G, that a particular node n_i in the tree has a particular nucleotide X in the base pair position of interest. X takes the value A, T, C, or G, representing the four nucleotides in DNA.

The likelihood distribution can be calculated for each node. For the leafs of the tree, which represent known samples from the dataset D, this probability is either one or zero. For example, $L_{n_7}(A) = 1$ because A is the actual value of n_7. For the same reason, $L_{n_7}(T) = 0$, since the actual value of n_7 is not T. At each interior node of the tree n_i, with child nodes n_j and n_k, for n_i to have value X implies that the nucleotide at n_i mutates to the values of each of its children during the interval that separates the parent from each child. So, for example, for n_3 to hold a value of A implies that an A mutated to the value of n_4 (or potentially did not mutate at all, in the case that n_4 is A) in 9.75 time units and that, independently, an A mutated to the value of n_8 (G) in 16.25 time units. However, in the general case, the child nodes will not have a

definite value. The child nodes will themselves have probability distributions across the four nucleotides ($L_j(X)$ and $L_k(X)$ for n_j and n_k, respectively.) So then

$$L_{n_i}(X) = \Big(\sum_Y P_{X,Y}(t_{i,j})L_{n_j}(Y)\Big)\Big(\sum_Y P_{X,Y}(t_{i,k})L_{n_k}(Y)\Big), \tag{19}$$

where Y is iterated across all four nucleotides in each summation, $t_{i,j}$ is the length of time in the interval between n_i and n_j, and $P_{X,Y}(t)$ is the probability that nucleotide X will mutate to nucleotide Y in time t. This probability is found[7] to be

$$P_{X,Y}(t) = e^{-ut}\delta_{X,Y} + (1-e^{-ut})\pi_Y, \tag{20}$$

where $\delta_{X,Y}$ is one if X and Y are the same and zero otherwise, u is a rate of mutation per measure of time, and π_Y is the prior probability of any given nucleotide being Y. The distribution π can be approximated by the relative frequency of each nucleotide in all the sampling data, since the biochemical causes of these frequencies are largely invariant.[7]. Equation 19 can be calculated for each node of the tree in the course of a post-order traversal. This will result in the likelihood of each possible nucleotide for the root. The term $L_{n_1}(A)$, for example, would be the probability that the given genealogical tree G would have produced the genealogical data at the base pair of interest, given that the root had an A at that base pair position. The overall likelihood of the genealogical tree G, for a particular base pair, can then be calculated as a scalar product of these likelihoods with the prior probability of each nucleotide:

$$L^i(G) = \sum_X \pi_X L_{n_1}(X). \tag{21}$$

So for each base pair position i, the term $L^i(G)$ is the probability that G would have resulted in the data D that were observed at i. The model described in [7] assumes that the mutation of each base pair position is independent, so adding the sequence dataset D as a variable term, the likelihood can be formally taken to be

$$P(D|G) = L(G) = \sum_i L^i(G). \tag{22}$$

It is important to note that the term u in Equation 20 is only relevant in relation to t, which is unit-less. The value of u is therefore independent from the θ parameter, and

$$P(D|G,\theta) = P(D|G). \tag{23}$$

The values calculated in Equation 18 and Equation 23 are crucial to the class of algorithm known as the coalescent genealogy sampler.

2.5 Coalescent Genealogy Samplers

A *coalescent genealogy sampler* is a tool used by population geneticists for estimating a variety of parameters of a population. An example of a population parameter that may be estimated is θ which, as defined in the previous section, is the product of the mutation rate and effective population size. The parameter θ is the motivating parameter for the development of coalescent genealogy samplers[7], and will be used to develop the concepts in this section.

Coalescent genealogy samplers employ MCMC methods to produce an estimate of θ given a sampling of sequence data, D, from the population being examined. Recall that MCMC methods involve constructing a Markov chain that samples a particular target distribution of a state space, then using the Monte Carlo method to approximate the expected value of some statistic of the state space. In this case, the state space is the set of possible genealogical trees with the sequence data D at the tips of the trees. The target distribution that is being sampled out of the state space by the Markov chain is the posterior probability of G, $P(G|D,\theta_0)$, where θ_0 is some initial guess at the true value of θ. The value θ_0 is also called the *driving value* of θ. The method of sampling out of this target distribution is implementation-specific, but it can be done with variations on the Metropolis-Hastings algorithm discussed in Section 2.3.[16]

The result of the sampling is a set $\{G\}$ of genealogies. Rather than a scalar value for the sample statistic, the Monte Carlo method is computing a function of the likelihood for arbitrary θ to produce the sample genealogy as a ratio to the likelihood of the driving value θ_0. It is should be anticipated that there are values of θ which are more likely than θ_0 to have produced $\{G\}$, despite the generation $\{G\}$ being driven by θ_0. This is because the target distri-

bution of $P(G|D,\theta)$ incorporates information from the sequencing data D. Even though, as noted earlier, it is the case that D can be determined independently of θ, there are still distributions of possible genealogies that are more compatible with D. The different possible distributions of $\{G\}$ are *not* independent of the value of θ.

Call the function which measures the ratio of the likelihood of an arbitrary θ value to that of θ_0 a relative likelihood function. The likelihood of θ to produce G and D can be factored as follows:

$$P(D,G|\theta) = \frac{P(D,G,\theta)}{P(\theta)} = \frac{P(D|G,\theta)P(G,\theta)}{P(\theta)} = \frac{P(D|G)P(G|\theta)P(\theta)}{P(\theta)} = P(D|G)P(G|\theta). \tag{24}$$

So then define the relative likelihood function[7] L_G such that

$$L_G(\theta) = \frac{P(D,G|\theta)}{P(D,G|\theta_0)} = \frac{P(D|G)P(G|\theta)}{P(D|G)P(G|\theta_0)} = \frac{P(G|\theta)}{P(G|\theta_0)}. \tag{25}$$

Note that the term $P(G|\theta)$ was defined in Equation 18. The output of the MCMC approximation is the unweighted average of Equation 25 across the entire set of sample genealogies $\{G\}$ and this is an approximation of the relative likelihood of an arbitrary value of θ:

$$L(\theta) = \frac{1}{N}\sum_G L_G(\theta) = \frac{1}{N}\sum_G \frac{P(G|\theta)}{P(G|\theta_0)}. \tag{26}$$

The resulting approximation $L(\theta)$ can be used to construct a likelihood curve, which is a graph of how much more likely a possible θ is to have produced the samples in G than the initial assumed value, θ_0. Such a curve can be seen in Figure 5, which shows an approximation of the relative likelihood curve, where the true θ parameter value for the population is 1.0 and the driving value of θ_0 was 0.01. It can be seen that values of θ close to 1.0 have higher likelihoods relative to θ_0. The maximum value of this curve should be taken to be the *maximum likelihood estimation* (MLE) of the value of θ. Assuming that the curve is continuous and differentiable, the MLE be found by performing an iterative gradient ascent. If desired, the initial presumed θ_0 can be replaced by the MLE of θ and the entire sampling process can be repeated to find a new curve and derive a new estimate of θ. This can continue for some

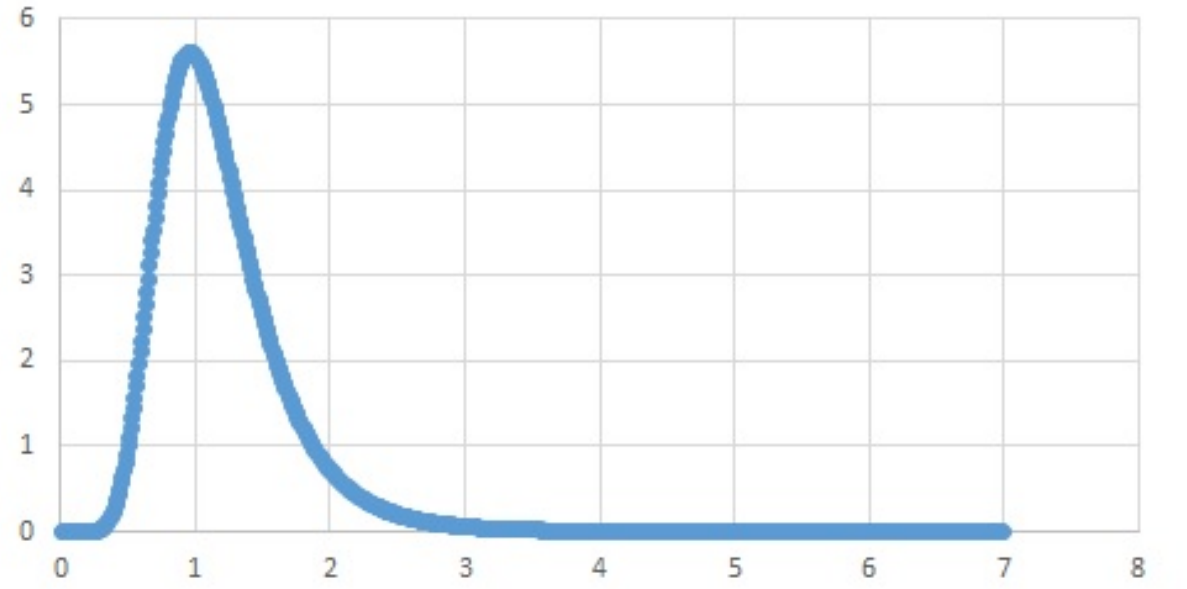

Figure 5: A likelihood curve with a true θ of 1.0 and a driving θ_0 of 0.01

fixed number of steps, or until the value of θ stabilizes in order to find a final estimate, which is the output of the method as a whole.

Coalescent genealogy samplers are applicable to a large diversity of studies and are able to provide significant insights into sampled populations.[16] The same method can be used to estimate parameters other than θ, provided a suitable sampler algorithms and likelihood functions can be computed. These versatile tools are an excellent application of computation to answer questions in science.

3 Problem Statement

Complicating the use of coalescent genealogy samplers is the fact that these types of algorithms are extremely computationally intensive, and as a result this type of analysis can be very time-consuming[16]. This is primarily due to the nature of the calculation of the likelihood values of individual genealogies. Recall from Section 2.3 that the calculation of the $P(D|G)$ term requires a post-order traversal of the genealogical tree for each base pair position in the sequence data. That term must be calculated in the process of sampling each genealogical tree, as will be explored in Section 4.2. This means that the execution time of the algorithm scales linearly with sequence length, sequence count, the number of genealogies sampled for each iteration of the likelihood maximization, and the number of iterations of the likelihood maximization that occurs. Large-scale multi-parameter analysis with a large amount of sequence data can potentially take on the order of weeks to complete with currently available coalescent genealogy sampling packages.

Because of this tremendous computational cost, it seems desirable to apply parallel computing to this problem at the greatest possible scale. Efficient application of parallelism would potentially enable the use of available computing resources to produce more accurate results in a much shorter time, reducing the amount of time devoted to analysis in many studies of population genetics. However, the nature of the Metropolis-Hastings algorithm at the heart of the sampler makes parallelization difficult [4]. Recall from Section 2.3 that a principle feature of a Markov chain is that the probability of transitioning to a given state depends solely upon the current state. This means that the current state must be known before the probabilistic distribution of the successor state can be determined. In other words, there is a serial dependency of each state transition upon the completion of the previous state transition. These dependencies make the parallelization of a Markov chain non-trivial.

A common work-around to the dependencies involved in a single Markov chain is to run multiple Markov chains in parallel [23] using the same set of transition probabilities — and hence the same process of exploring the state space — for each individual chain. The results are then aggregated to provide the final result. In the case of the coalescent genealogy sampler, the sets of sample genealogical trees would be collated into one aggregate set for cal-

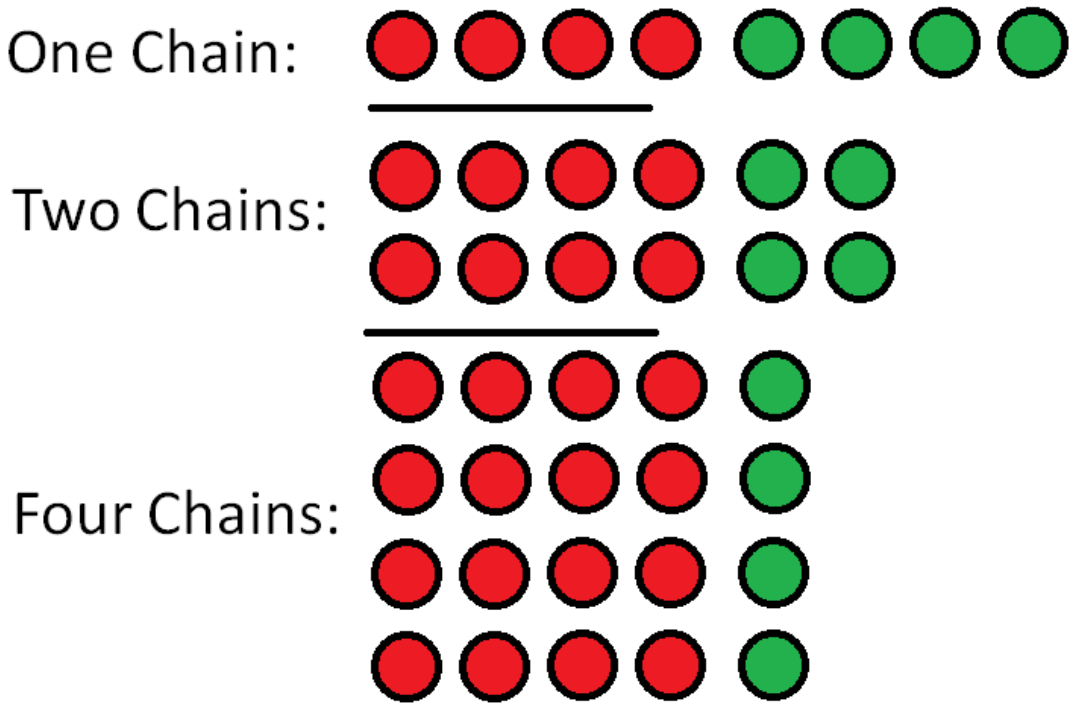

Figure 6: An illustration of the efficiency of combining multiple chains. Each chain must go through a burn-in period of 4 ($B = 4$) steps through the chain to ensure convergence. The total output of all the chains is 4 ($N = 4$) samples. The number of chains is equal to P. As the number of chains increases, the amount of computation time spent in the burn-in period dominates, with diminishing returns for time of execution.

culating the likelihood curve. This method, with some additional exchange of information between chains, is proposed in [25]. However, this approach is increasingly less efficient at high degrees of parallelism due to the burn-in period. Each Markov chain must complete its own burn-in before non-biased sampling of the target distribution can begin. As mentioned in Section 2.3, the optimal length of this period can be difficult to assess. As a point of reference, the current implementation of one coalescent genealogy sampler software package uses a fixed burn-in period. Suppose a fixed burn-in period of B transitions is necessary to be certain the Markov chain has converged to its stationary distribution. These B transitions will be discarded for sampling purposes, but they are a necessary step every time a chain is run. If N samples are to be taken from the target distribution, then a single chain will need to perform $B+N$ transitions in order to acquire the desired N samples. With P processors and one Markov chain per processor, each processor will need to calculate $B+\frac{N}{P}$ transitions. An example of this is illustrated in Figure 6. As P increases, B (which is the serial component of the execution) will dominate the runtime equation in accordance with Amdahl's law [1], which observes that as parallelism increases, the remaining serial component will dominate the computation:

$$\lim_{P\to\infty} B+\frac{N}{P} = B. \tag{27}$$

This means that increasing parallelism will have diminishing returns. Hence, although the method of aggregating multiple chains will result in an improved runtime for low-order parallelism, it will not efficiently scale on highly-parallel computing clusters.

In order to make efficient, high-order parallelism of coalescent genealogy samplers possible, a more sophisticated method of parallelization is required. A method for doing so is described here, with implementation details and results.

4 Method

The method that is be explored in this section is the application of a variant of the Generalized Metropolis-Hastings algorithm[4] to the coalescent genealogy sampler that is described in the seminal paper[18] on the LAMARC software package. This is done in order to achieve scalable parallelism of modern coalescent genealogy samplers. The parallel platform used is the general-purpose graphics processing unit (GPGPU), utilizing the CUDA programming framework.[21]

4.1 Generalized Metropolis-Hastings

This method involves an application of a variant of the recently-discovered *Generalized Metropolis-Hastings* algorithm.[4] Generalized Metropolis-Hastings, or *Calderhead's method*, differs from standard Metropolis-Hastings in that, at each iteration of the algorithm, it makes multiple proposals instead of a single proposal. That is, an implementation of Calderhead's method requires some proposal mechanism, or *proposal kernel*. This proposal kernel produces an ordered set of N new candidate states. As in the case of standard Metropolis-Hastings, there is a current state which determines the probability distribution out of which the set of proposals is drawn, but unlike standard Metropolis-Hastings, the space on which this probability distribution is defined is the set of N-tuple vectors of the states of the state space.

Once a set of N proposals has been generated, a transition matrix A can be calculated for the $N+1$ possible successor states (the N proposed states plus the current state.) The transition probabilities of A are determined in such a way that the flow from one state to another achieves equilibrium with the two states occurring at the same relative frequency as in the target distribution. Therefore, the stationary distribution of a Markov chain defined by the transition probabilities of A will sample from the set of proposals in a way that is proportional to those proposals probabilities in the target distribution.

After the transition matrix A has been calculated, it is possible to sample from the stationary distribution of the Markov chain defined by A directly. This sampling can be done some arbitrary number of times, and the samples generated are the output of the sampler. These

samples are proportional to the target distribution. The last sample taken using A will be used to generate a new set of proposals, and the process will repeat until the total number of samples generated is the desired output of the chain, or some other termination condition has been met.

Calderhead provides an overview of the method[4]; a modified version appears in Algorithm 1. The notation of $\tilde{x}_i$ represents the ith member of the proposal set, while x_i represents the ith sample taken by the method as a whole. Note that "the stationary distribution of I conditioned on the set of proposals and the state used to generate those proposals" is just the stationary distribution of the Markov chain defined by the transition matrix A. The stationary distribution of this Markov chain $P(\tilde{x}_i)$, for some member $\tilde{x}_i$ of the set of proposals, is proportional to $\pi_{\tilde{x}_j} K(\tilde{x}_j, \tilde{\mathbf{x}}_{\setminus j})$, which is the product of the density of the state in the target distribution with the probability of that state generating the rest of the current proposal set via the proposal kernel K.

Algorithm 1 Calderhead's Method

procedure GENERALIZED METROPOLIS-HASTINGS
    Initialize starting atomic state $\tilde{x}_1$, $i = 1$, and counter $n = 0$.
    for each iteration **do**
        Update the set of proposals $\tilde{\mathbf{x}}_{\setminus i}$ by drawing N new points from a proposal kernel, call this $K(\tilde{x}_i, \cdot)$
        Calculate the stationary distribution of I conditioned on the set of proposals and the state used to generate those proposals.
        for $m = 1$ to N **do**
            Sample i from the stationary distribution of I, setting the sample $x_n + m = \tilde{x}_i$.
        end for
        $n = n + N$
    end for
end procedure

Calderhead's method is called *Generalized* Metropolis-Hastings because, when N (the number of generated proposals) is one, the method is no different from the standard Metropolis-Hastings algorithm. Although the presence of a secondary transition matrix that guides the transition probability during the sampling stage may seem like a departure from the requirement that the transition probabilities of a Markov chain are based *solely* on the current state (and not on any other variables), this is not the case. Calderhead's method makes use of an *auxiliary variable* [2], I, which serves as the index of the set of proposals generated by the

proposal kernel. Formally, the states being traversed by the method are defined by an ordered set of $N+1$ "states" of the original problem *plus* some value of I. The method as a whole is a *mixture* [2] of two different types of transitions. The first transition is the proposal stage, which changes the set of proposals. The second is the sampling stage, which changes the value of I. Additionally, at each sampling step, the sampling output is taken to be the member of the proposal set indexed by I. It is important to realize that this method is not truly transitioning from proposal to proposal, but is rather transitioning between more complex states and translating these more complex states into single proposals at each sampling step.

This method provides several opportunities for improved parallelism over a standard implementation of Metropolis-Hastings. First, the method of proposal generation can be parallelized. At the step in which the N proposals are generated, the only state information required is that of the current state, $\tilde{x}_{\setminus I}$.

This method has several points of more efficient parallelism that the standard Metropolis-Hastings method. First and foremost, the proposal kernel is dependent only upon the current state, $\tilde{x}_{\setminus I}$. While that is a liability to parallelism in standard Metropolis-Hastings, in Calderhead's method each proposal is able to be generated independently from all others. This allows each processor to separately generate a proposal. In other words, there is no interdependency between the processes generating the individual proposals. Additionally, most of the computation involved in the construction of the transition matrix can be done in parallel. Since the matrix is $(N+1)\times(N+1)$ in size, each processor can populate one row of the matrix. For coalescent genealogy samplers, these two phases — proposal and sampling — account for the vast bulk of the computation. This converts a necessarily serial method to one that is very readily parallelizable. Note also that there is no distinction between the parallelism applied to the burn-in phase and the sampling phase. In contrast to other[25] methods, there is not a necessarily-sequential burn-in component.

Applying Calderhead's method to coalescent genealogy samplers requires developing a proposal kernel that can generate multiple proposals which can be sampled. Recall that the stationary distribution of the Markov chain defined by transition matrix A is proportional to $\pi_{\tilde{x}_j} K(\tilde{x}_j, \tilde{\mathbf{x}}_{\setminus j})$ for any $\tilde{x}_j$ in the proposal set. This means that for it to be possible for a particu-

lar proposal $\tilde{x}_j$ to be sampled, it must be *possible* for the multiple proposal kernel to propose the rest of the proposal, to account for the situation that $\tilde{x}_j$ is the current state at the start of the proposal phase. In other words, for $\tilde{x}_j$ to possibly be sampled, it must be the case that $K(\tilde{x}_j, \tilde{\mathbf{x}}_{\setminus j}) > 0$.

4.2 The LAMARC Package

The method described here is a modification of the software package LAMARC (Likelihood Analysis with Metropolis Algorithm using Random Coalescence), which is described in the seminal paper on coalescent genealogy samplers.[18] Although the contemporary LAMARC program supports estimating several different parameters, the method in this thesis will only be relevant to the θ product described in Section 2.4. This approximation is found by iteratively performing an Expectation-Maximization (EM) algorithm for the likelihood of θ through Markov sampling of genealogies as an implementation of the algorithm described in Section 2.5.

The conventional version of LAMARC implements a standard Metropolis-Hastings algorithm for the Expectation-generation step of the EM algorithm, and that implementation will be modified for the purpose of this research. The standard proposal mechanism of LAMARC is to target one of the non-root interior nodes (the *target* node) at random from the current genealogical tree and resimulate the neighborhood around that node in such a way that proposals are selected from the distribution $P(G|\theta)$. See Figure 7 for an example of a genealogical tree with the target, parent, and children labeled. Recall that each interior node represents a coalescent event. Resimulating the neighborhood consists of replacing the two deleted nodes, and possibly reordering the children of those nodes, in accordance with population genetics and coalescent theory.[15]

Resimulating proportionally to $P(G|\theta)$ is achieved by treating the genealogical tree as a series of time intervals in order to determine in which time intervals the coalescent events can legally be placed probabilistically, and then placing them probabilistically inside those intervals. Figure 8 shows the tree from Figure 7 that is being resimulated. The lineages that have been removed and have to be resimulated are called *active lineages*. At the beginning of

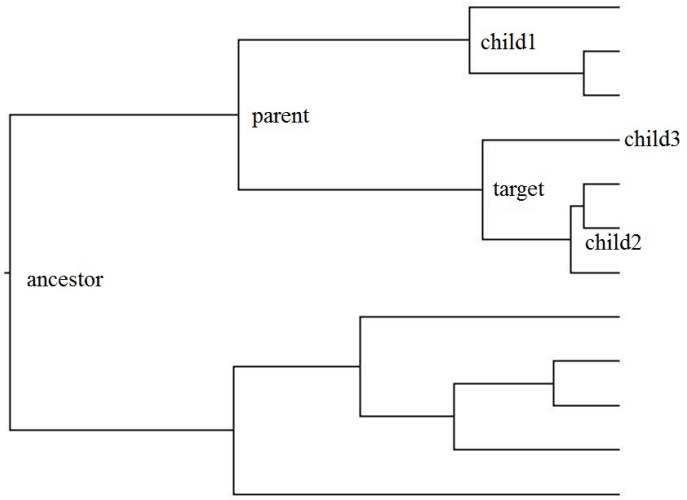

Figure 7: A tree with the neighborhood of resimulation labeled

the region being resimulated, there is a single active lineage. This lineage descends from the *ancestor* node, which is the parent of the parent of the targeted node. The first task in resimulation is to establish where coalescent events occurred. At each coalescent event, one active lineage splits into two. The earliest a coalescent event could occur is immediately following the ancestor. The times of coalescent events are bound by the location of the children in the region being resimulated. At each child, an active lineage terminates, so coalescent events must occur before the children that descend from them. Any interval that could contain a coalescent event is a *feasible intervals*. In Figure 8, there are six feasible intervals. The significance of these as defined intervals is that the same number of lineages are present for the duration of the interval, which is relevant to the resimulation of the neighborhood.

The method of finding the interval in which the active lineages coalesce occur (i.e. the interval contains a coalescent event) is to calculate the probability of zero, one, or two active lineages coalescing for each interval, and then work backwards from the knowledge that there is exactly one active lineage at the end of the last feasible interval for coalescent placement. Suppose that higher-numbered intervals are chronologically later than lower-numbered intervals, so that interval $i+1$ is the interval immediately following interval i. Further, suppose that $P_i(n)$ is the probability that there are n active lineages at the start of interval i and that $S_{i,j}(t)$ is the probability of starting an interval of length t with i active lineages, but finishing with j active lineages. As an example, $S_{2,3}(1.0)$ is the probability of a single coalescent event happening in an interval of time length 1.0 which ends with three active lineages. Knowing

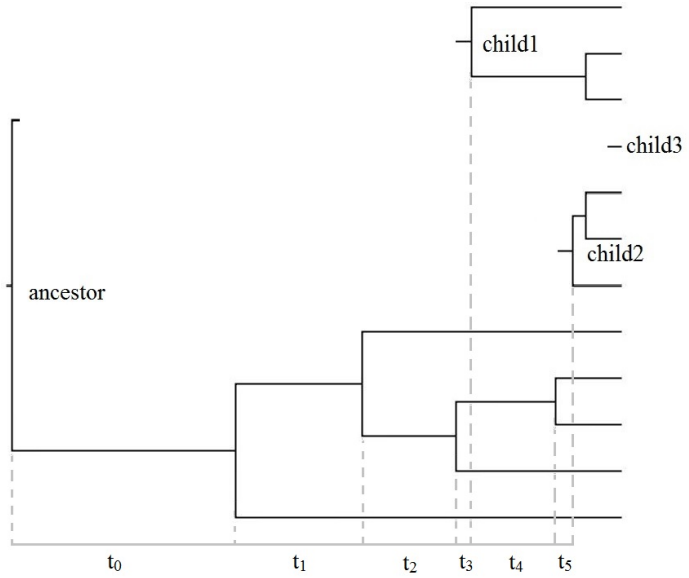

Figure 8: A tree with the neighborhood of resimulation deleted, intervals marked

that the number of active lineages at the end of any feasible interval is at least one and at most three, it is possible to determine the values of $P_i(n)$ with the length of interval i, call this t_i, and the values of $P_{i+1}(n), n \in 1,2,3$ as follows:

$$P_i(1) = P_{i+1}(1)S_{1,1}(t_i) + P_{i+1}(2)S_{1,2}(t_i) + P_{i+1}(3)S_{1,3}(t_i)$$
$$P_i(2) = P_{i+1}(2)S_{2,2}(t_i) + P_{i+1}(3)S_{2,3}(t_i)$$
$$P_i(3) = P_{i+1}(3)S_{3,3}(t_i).$$

The probability function $S_{i,j}(t)$ can be derived from Kingman's coalescent theory [15] based upon a constant chance of coalescence of two active lineages, integrated over the length of the interval. This constant chance of coalescence is a function of the number of active lineages, the number of inactive lineages (lineages that are not being resimulated) and the parameter θ. Given the way intervals have been defined, there are a fixed number of inactive lineages throughout the interval. These probabilities can be used to continue backwards through the feasible intervals, populating the values of $P_i(n)$, until the ancestor node is reached.

Once the probabilities of $P_i(n)$ have been populated for all feasible intervals, the genealogical tree can then be resimulated starting from the earliest feasible interval, with the knowledge that there is exactly one active lineage at the beginning of that interval. This is accom-

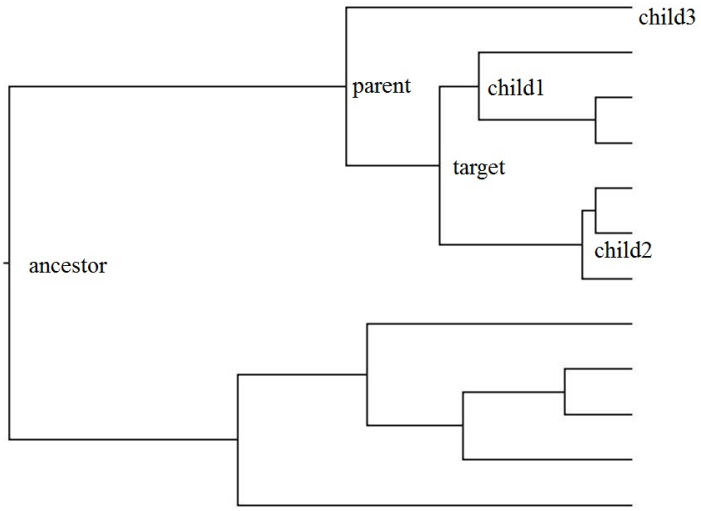

Figure 9: A possible resimulation of Figure 7

plished by a walk forward in time, probabilistically deciding if the next interval has zero, one, or two coalescent events weighted by the values of $P_i(n)$. Once the interval in which all coalescent events will occur has been selected, the probabilistic distribution of placement within an interval is selected by treating $S_{i,j}(t)$ as a cumulative distribution function of the density of the coalescent. Additionally, the proposal may rearrange the children of the original target and its parent in order to change the structure of the overall tree.[18]

The end result is a proposal which is a candidate successor genealogy. It is a modification at one interval of the current genealogical tree. See Figure 9 for an example of how Figure 7 could have been resimulated. Notice that the children have been reshuffled, changing the structure of the tree, not just the timing of the coalescent events. In this way, proposals are generated out a distribution that is proportional to $P(G|\theta)$, where G is the proposed genealogy. Following the Metropolis-Hastings algorithm[13], the acceptance ratio necessary to sample from the posterior distribution $P(G|D,\theta)$ is

$$r = \frac{P(G|D,\theta)P(G_0|\theta)}{P(G_0|D,\theta)P(G|\theta)} = \frac{\left(\frac{P(D|G_0)P(G|\theta)}{P(D|\theta)}\right)P(G_0|\theta)}{\left(\frac{P(D|G)P(G_0|\theta)}{P(D|\theta)}\right)P(G|\theta)} = \frac{P(D|G)}{P(D|G_0)} \tag{28}$$

The proposal is accepted with probability $min(1,r)$.

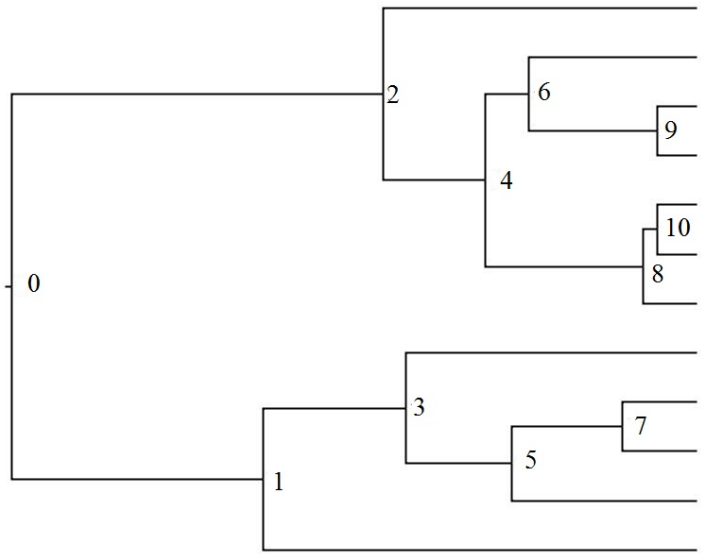

Figure 10: A tree with node labels

4.3 Modifying LAMARC for Calderhead's Algorithm

There is a challenge in applying Calderhead's algorithm using LAMARC's proposal mechanism. If two genealogies vary by more than one neighborhood, they cannot mutually propose each other. This means that Calderhead's method cannot be applied directly by simply sampling N times using the standard LAMARC mechanism, as this has a high probability of producing proposals that cannot mutually propose some other member of the proposal set. For example, consider the tree in Figure 10. If one proposal, $\tilde{x}_i$, is generated by resimulating the neighborhood around target node 6 and another proposal in the proposal set, $\tilde{x}_j$, is generated by resimulating the neighborhood around any other node, then the two proposals will vary by more than one neighborhood. There is no way that the standard proposal mechanism of LAMARC will be able to propose $\tilde{x}_j$ as a successor to $\tilde{x}_i$, or vice versa. In fact, any other proposal in the set will vary by more than one neighborhood from $\tilde{x}_i$, $\tilde{x}_j$, or both. This means that if *any* two proposals vary by more than one neighborhood, then no member of the proposal set could mutually propose the rest of the proposal set other than the original generator of the set. Hence, no other member of the proposal set could be sampled, which is pathological.

To avoid this problem, the method devised by research project is to use a modified proposal mechanism. This proposal mechanism maintains an auxiliary variable. This variable, call it φ, is sampled from a uniform distribution of $1 : N$, where N is the number of interior

nodes in the genealogical tree. The value of φ is sampled prior to each proposal set being generated, and determines which neighborhood is targeted for resimulation.

Because φ is picked from a uniform distribution (without being informed by any of the state information of the Markov chain) it is trivially invariant. The impact of this variable is to reduce to zero the probability of a generator state generating a set of proposals where any two proposals vary by more than one neighborhood. Any remaining set has its probability of being proposed reduced by a factor of N, however the probability distribution of these sets remains proportional to the distribution in the original LAMARC proposal mechanism. This means that the target distribution of the method remains the same, preserving correctness. This modification ensures that all proposals will be able to mutually propose the entire set, guaranteeing a non-zero sampling probability during the application of Calderhead's method.
Because φ is picked from a uniform distribution (without being informed by any of the state information of the Markov chain) it is trivially invariant. The impact of this variable is to reduce to zero the probability of a generator state generating a set of proposals where any two proposals vary by more than one neighborhood. Any remaining set has its probability of being proposed reduced by a factor of N, however the probability distribution of these sets remains proportional to the distribution in the original LAMARC proposal mechanism. This means that the target distribution of the method remains the same, preserving correctness. This modification ensures that all proposals will be able to mutually propose the entire set, guaranteeing a non-zero sampling probability during the application of Calderhead's method.

Instead of a single acceptance ratio, Calderhead's method computes a stationary distribution on the proposal set, and samples states directly from that distribution. Recall that, in terms of coalescent genealogy samplers, the distribution is proportional to $\pi_{\tilde{G}_i} K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i})$ for any $\tilde{G}_i$ in the proposal set, where $\pi_{\tilde{G}_i}$ is the posterior probability of the proposal and $K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i})$ is the probability of $\tilde{G}_i$ mututally proposing the rest of the proposal set. These terms are calculated as follows:

$$\pi_{\tilde{G}_i} = P(\tilde{G}_i|D,\theta) = \frac{P(D|\tilde{G}_i)P(\tilde{G}_i|\theta)}{P(D|\theta)} \tag{29}$$

and

$$K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i}) = \prod_{j \neq i}^{N+1} P(\tilde{G}_j|\theta) = \Big(\frac{P(\tilde{G}_i|\theta)}{P(\tilde{G}_i|\theta)}\Big) \prod_{j \neq i}^{N+1} P(\tilde{G}_j|\theta) = \frac{\prod_j^{N+1} P(\tilde{G}_j|\theta)}{P(\tilde{G}_i|\theta)} \propto \frac{1}{P(\tilde{G}_i|\theta)}. \quad (30)$$

So then

$$\pi_{\tilde{G}_i} K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i}) \propto \Big(\frac{P(D|\tilde{G}_i)P(\tilde{G}_i|\theta)}{P(D|\theta)}\Big) \frac{1}{P(\tilde{G}_i|\theta)} = \frac{P(D|\tilde{G}_i)}{P(D|\theta)} \propto P(D|\tilde{G}_i) \quad (31)$$

This distribution is easily sampled from, given a discrete set of proposals by sampling a real number, x, uniformly from the interval $(0, \sum_{i=1}^{N+1} P(D|\tilde{G}_i))$, and iterating through the proposals until reaching the lowest j such that $\sum_{i=1}^{j} P(D|\tilde{G}_i) \geq x$. The hidden variable I then takes value j, and the Markov chain samples the genealogy $\tilde{G}_j$. This sampling occurs an arbitrary number of times before generating a new sample set out of the distribution $K(\tilde{G}_i, \tilde{\mathbf{G}}_{\setminus i})$.

4.4 General-Purpose Graphical Processing

The platform chosen for applying Calderhead's method to the coalescent genealogy sampler is the Nvidia CUDA platform[21]. CUDA is a *General-Purpose Graphical Processing Unit* (GPGPU) platform, which repurposes graphics hardware designed for the massive parallelism of graphics rendering for regular computation. This allows a very high throughput of computation with a relatively small form factor. A CUDA graphics card makes hundreds or thousands of processing units available for simultaneous execution. In the CUDA paradigm, there are two different processing contexts: the *host*, which is the CPU executing as a traditional computing environment to initialize memory and run programs in the second context, which is the graphics card (or *device*.)

CUDA is a single instruction, multiple data (SIMD) architecture, which means that every computation unit in the same thread group, or *warp*, must be running the same instruction at the same time, but may be running that instruction upon different data. The programs that are executed on the graphics device itself are called *CUDA kernels*. The SIMD architecture introduces some constraints into the design of CUDA kernels. For example, the efficiency of execution is maximized when there is minimal conditional branching in a program, or when all parallel threads of execution take the same branch. This is because different paths of execu-

tion must be executed serially, rather than in parallel.

There are several generations of the CUDA platform, reflected in the versioning of two separate components. The first component is the hardware. The version of hardware is referred to as the *compute* version. Each graphics card has a compute version, which is associated with a different architecture, scale, and capabilities. The graphics card used for this research had compute version 3.5, which is in the Kepler family. The second component of the versioning is the API, or *toolkit* version. Different toolkits provide support for different compute versions and feature sets. The toolkit version used for this project was 7.5.

The availability of compute version 3.5 hardware enabled the use of dynamic parallelism. This feature is essentially the ability to launch CUDA kernels from within other CUDA kernels. This allows the likelihood calculation for proposal genealogical trees to be launched at the end of the proposal generation process as individually parallelized CUDA kernels. Launching the likelihood calculations from inside the proposal kernel simplifies program flow and design, removing the need for either a more complex reduction step if the likelihoods of all proposals were launched from a single CUDA kernel at the host level or the implementation of a serially executed loop launching likelihood-calculating CUDA kernels for each proposal. The flow of these kernels will be explored in greater detail in Section 5.1.

Another advantage of using compute version 3.5 is access to the native shuffle operations. These are native, or *intrinsic* operations on the device itself for sharing information between the threads in a warp. Shuffle operations allow for much more efficient *reductions* of data. An example of a reduction that would be improved by the shuffle operations would be the summation of a set of numbers, wherein each execution thread owned a number. Shuffle operations allow each thread in a warp to share a value with another thread in the same warp as an atomic operation. This negates the need for multiple explicit synchronizations of a larger group of threads known as a *block* over the course of a reduction, which is suboptimal due to the specific implementation of the block synchronization operation. Additionally, using shuffles requires less shared memory be allocated between threads, as threads can share values from their registers with other threads in the same warp directly, rather than needing to copy these values to intermediate shared memory.

Choosing a toolkit version greater than 6.0 makes it possible to use the *unified memory addressing* (UMA) feature. This allows memory to be commonly addressed and accessed in both host code and device code (i.e. CUDA kernels.) The advantage of this is that an object declared as unified memory does not need to be explicitly copied between the host and the device as calls to the unifed, or *managed*, address space automatically trigger a synchronization of the two memories. Previous to the existence of unified memory, copying complex data structures was very difficult, since any memory references would have to be reallocated to locally make sense in the different memory space when moving from device to host or host to device.

5 Implementation

In order to test the effectiveness of the method described, a proof-of-concept implementation program was created.

5.1 Program Flow

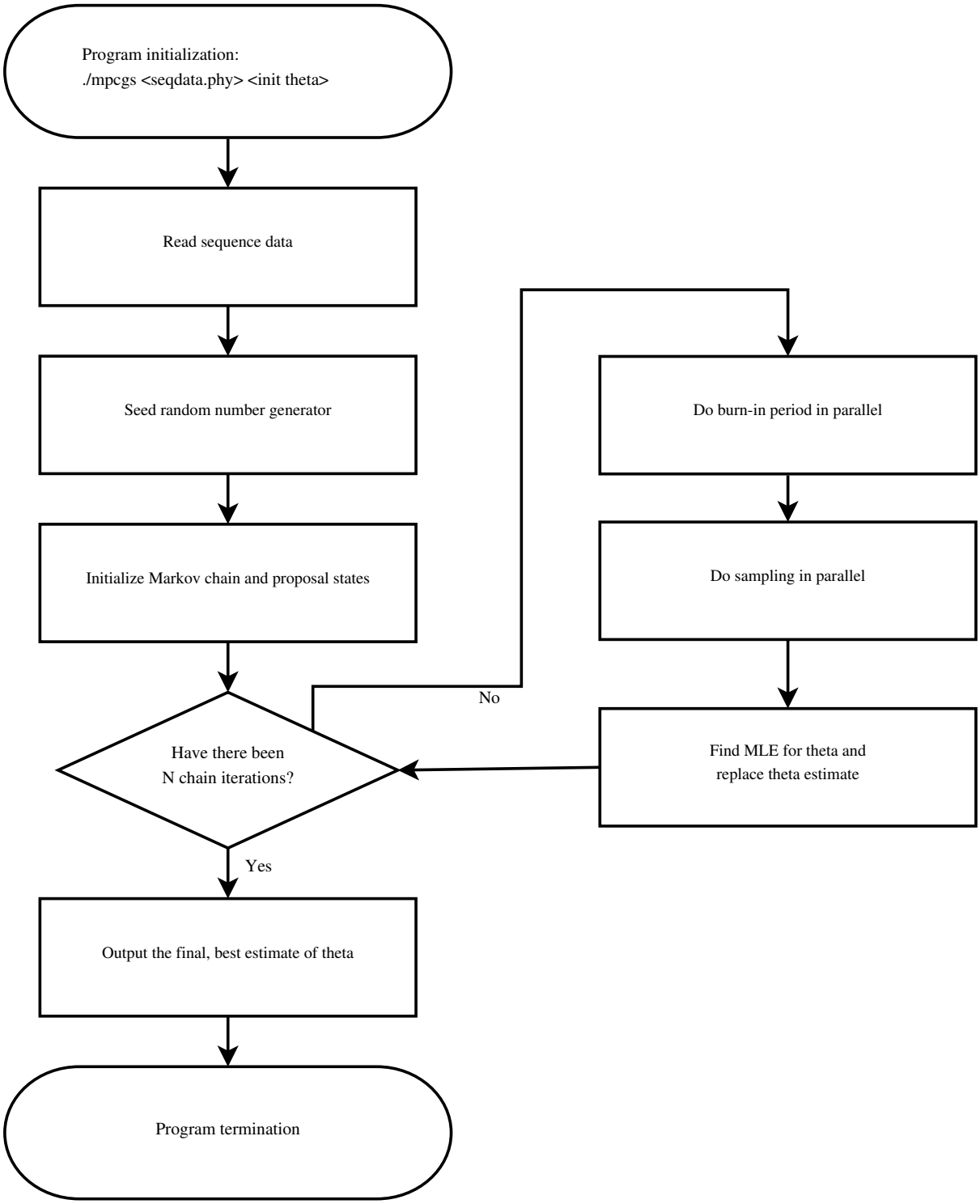

Figure 11: The overall flow of the program. The main loop will be run N times, to perform N iterations of the Expectation-Maximization algorithm. N is statically defined.

The main flow of the program can be seen in Figure 11. This section gives implementation details of this figure, expanding as necessary.

5.1.1 Program Entry

The proof-of-concept that was developed for this research is called mpcgs(multi-proposal coalescent genealogy sampler.) The command-line arguments that mpcgs receives at initialization are the name of a file that contains the formatted sequence data and an initial estimate of θ (θ_0.) The sequence data are expected to be in the PHYLIP genealogical data format[8], in which the first line provides the number of samples and the length of the samples. Each successive line leads with a fixed-length name of the sample followed by the sequence data. For genealogical data, each base-pair is represented by a letter corresponding to the associated nucleotide: A for adenine, C for cytosine, G for guanine, and T for thymine. The sequence data pulled from the file specified in the first command-line argument constitute the D term as described in earlier sections. There are no constraints on the initial estimate of θ that is provided as the second command-line argument, beyond being positive: the method as a whole is designed to be insensitive to the initial driving value of θ.

5.1.2 Pseudo-Random Number Generator

There are two *pseudo-random number generators* (PRNGs) utilized in the program. The first is MT19937, a common variant of Mersenne Twister.[19]. The state for this PRNG is maintained on the host, and it is for random number generation by the CPU. This PRNG is used primarily to sample the auxiliary value φ from a discrete uniform distribution. However, the PRNG used by the host cannot be used by the CUDA kernels to generate random numbers since the state is held in the memory of the host and is inaccessible to them at runtime. Additionally, the standard deployments of MT19937 are not optimized for use by a GPGPU. It is therefore necessary to have a second PRNG, with its state held in the memory of the graphics card. An implementation exists of Mersenne Twister for CUDA, MTGP32, based upon [24]. This implementation maintains state for up to 256 separate threads simultaneously, and is thread-aware when run inside a CUDA kernel. This means that calls from different threads

keep their state independently, with a goal of zero correlation between the numbers generated for different threads at the same point in execution.

5.1.3 Data Initialization

During the initialization stage, all memory that will be necessary over the course of the program's execution is preallocated, values that are invariant across the execution of the program are populated in device memory, the initial tree G_0 is generated and the term $P(D|G)$ is calculated.

The data structures that are preallocated include a place-holder proposal set for performing Calderhead's method. This set consists of $N+1$ genealogical tree structures, where N is the number of proposals that will be made in the proposal stage. The preallocated data structures also include space for M genealogies that have been reduced to an array of time-intervals, where M is the number of genealogies sampled for MLE calculation. Nothing more than the time intervals are stored for each sample, since the time intervals between the coalescent events are all that is necessary to calculate the $P(G|\theta)$ term; i.e. the structure of the genealogy is not needed. Additionally, since memory allocation inside a CUDA kernel is much less efficient than allocations performed remotely on the host, it is advantageous to allocate memory for intermediate calculations as "scratch" space at this stage, rather than to continually allocate and deallocate space at run-time.

The CUDA platform provides Read-Only memory specifically for constants that will not change over the lifetime of a CUDA kernel. This constant space is highly optimized for read access. For the coalescent genealogy sampler, the data that will not change over the lifetime of execution are the sequence data. It is therefore desirable that this information be stored in constant memory for improved performance. Constant memory has optimal performance when every thread in a warp is reading the same value from constant space at the same time. In all versions of the CUDA platform, there are 32 threads in each warp. There are four possible values that each base pair position in each sequence can take (A, C, G, or T), so each base-pair position in each sequence can be stored in two binary bits. As will be seen in Section 5.2.2, the likelihood calculation kernel will need each thread to have the value of a single

position in a single sequence, so an entire warp can be populated out of 64 bits of data. This means that each thread in the warp can read the same 8-byte value out of constant space in order to optimize the reading of sequence data in the kernel.

Following [18], mpcgs initializes the initial starting genealogy of the Markov chain, G_0, to be the UPGMA tree generated[8] by the distance between sequences in D. A UPGMA tree is built by clustering leafs and sub-trees according to some distance measure. The distance between individual sequences is taken to be the number of base pair positions that are different between the two sequences, while the distance between two subtrees is the arithmetic mean of the distances between all the leafs across the two trees. Shorter branch lengths mean closer relations. As a deviation from the standard application of UPGMA, the branch lengths are scaled by the assumed driving value of θ. The Markov chain is then seeded with G_0 as the generating genealogy, which accomplishes step 2 of Algorithm 1.

5.1.4 Sampling by Calderhead's Method

Once the data initialization stage is complete, the Markov chain can be run using Calderhead's method. Running the chain involves two distinct periods: the burn-in period and the sample generation period. Both of these periods are realized by repeated iteration of Algorithm 1, with the addition that φ is sampled prior to each proposal set generation and passed to the CUDA kernel (called the proposal kernel) that produces proposal genealogies. The result of the proposal kernel is the generation of a proposal set, each member of which has a non-zero probability of proposing the rest of the set, given the value of φ. The proposal kernel additionally calculates the terms of the stationary distribution that allows sampling of the proposal set. The flow of this process can be seen in Figure 12.

5.1.5 Maximum Likelihood Estimation

Once sampling is complete, the program searches for the value of θ that has the greatest relative likelihood, calculated using Equation 26. This is accomplished through an iterative gradient ascent of the relative likelihood curve, the basic method of which is shown in Algorithm 2. The value of $L(\theta)$ is computed using another CUDA kernel, the *posterior likelihood kernel*.

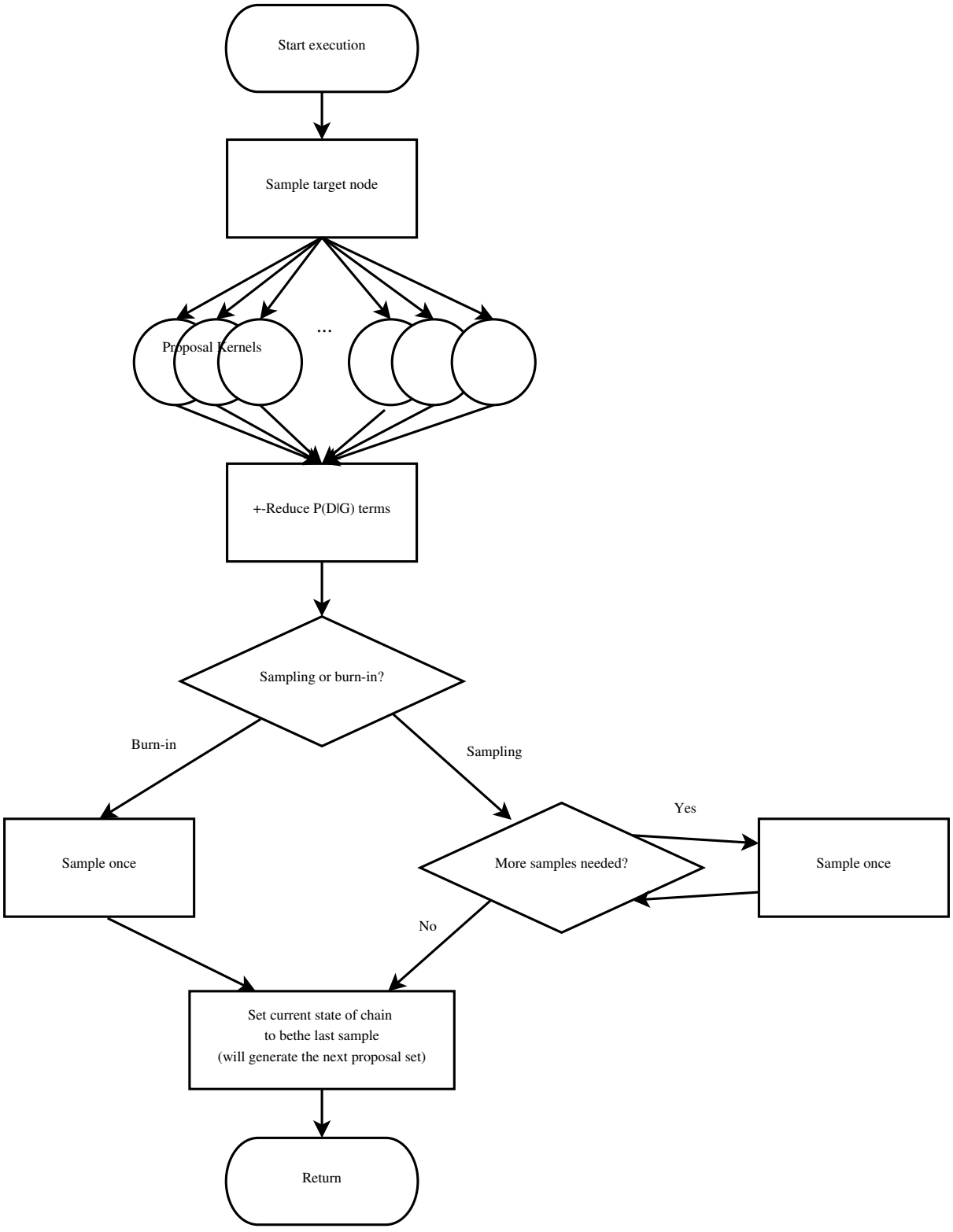

Figure 12: The flow of the multiple proposal and sampling mechanism

Algorithm 2 Gradient Ascent

set δ, ε very small
$\theta_{next} = \theta_0$
do
    $\theta = \theta_{next}$
    $\texttt{gradient} = \frac{L(\theta_{next}+\delta)-L(\theta_{next}-\delta)}{2\delta}$
    while $(L(\theta_{next}) - L(\theta_{next}+\texttt{gradient})) > \delta$ **or** $(\theta_{next}+\texttt{gradient}) < 0$ **do**
        $\texttt{gradient} = \texttt{gradient} \times \frac{1}{2}$
    end while
    $\theta_{next} = \theta_{next} + \texttt{gradient}$
while $|\theta - \theta_{next}| > \varepsilon$
$\theta_0 = \theta_{next}$

5.2 Kernels

The program described here implements three separate CUDA kernels: the proposal kernel, the data likelihood kernel, and the posterior likelihood kernel. The operation of all three will be described in this section.

5.2.1 Proposal Kernel

The proposal kernel implements an algorithm very similar to the original LAMARC[18] proposal method, as described in Section 4.3. The main difference in method is that the neighborhood targeted for resimulation is passed to the kernel as an argument. Each thread running the proposal kernel generates one proposal genealogy. Also, each thread running the proposal kernel has the responsibility of calculating the value of $P(D|\tilde{G}_i)$, where $\tilde{G}_i$ is the proposal generated by that thread.

Since each thread produces one proposal, the total size of the generated proposal set will be equal to the number of threads running the kernel in parallel. Each thread has an ID assigned by the graphics card, and that thread ID is taken to be the index of the generated proposal in the proposal set. The thread with an ID equal to that of the current generator for Calderhead's method is idle for most of the processing, as it does not have to produce a new proposal.

All random numbers are generated prior to any possible branching of the thread execution. That is because it is a necessary condition of the MTGP32 PRNG that all threads be executing

the same random number generation request at the same time, to avoid one thread overrunning the state of another thread. [21] The random numbers are generated prior to the resimulation of the target neighborhood, and are stored in space that was allocated during the data initialization step.

Once the proposal has been generated, each thread individually initiates an instance of the data likelihood kernel in order to determine the value of $P(D|\tilde{G}_i)$, which is the data likelihood of $\tilde{G}_i$. In order to guarantee parallel execution, a different stream of execution is initialized for each data likelihood kernel. Note that there are two layers of parallelism occurring: multiple individual proposal threads are each launching multiple child threads to calculate the data likelihood. This is an example of dynamic parallelism. Because of the large number of total threads across all kernel executions, the load of the data likelihood calculation as a whole is shared across all the processing units of the graphics card.

After the data likelihood has been calculated, an additive reduction is performed across all the proposal threads. This is done by using the warp shuffle operators to reduce each warp down to a single value, placing that value into shared memory, and then additively reducing the values in shared memory down to a single value, a step which occurs on a single thread. Although it may seem inefficient to serially reduce this value in a single thread, in practice the number of warps will be small enough that this is not a significant portion of the total computation.

The output of this kernel is a set of proposals, as well as a discrete weighted distribution on those proposals that can be readily sampled from in order to produce sample genealogies.

5.2.2 Data Likelihood Kernel

The data likelihood kernel calculates the value of $P(D|G)$ for a given genealogy G. The D term represents the sequence data, which will be constant throughout the execution of the program. This kernel implements the post-order traversal method described in Section 2.4. Each thread calculates the likelihood for a particular base pair position in the sequence data. In other words, each thread calculates $L^i(G) = \sum_X \pi_X L_{n_1}(X)$, where i is taken to be the thread ID.

In theory, it is only necessary to recalculate the likelihood of nodes of the tree that are the parents of branches that have changed due to the resimulation of the tree. If no aspect of a tree's structure or branch lengths has changed, then its data likelihood will not have changed either. However, in practice, the cost of uncached memory access on the CUDA platform means that it is computationally more efficient to simply recalculate the likelihood of every node in every tree for every proposal, as opposed to reading some of the values from memory. Memory accesses *are* required in order to find the sequence data at the tips of the tree. However, as mentioned in Section 5.1.3, this information is stored in constant space, which decreases access times substantially.

In order to keep all intermediate calculations in stack registers, the program is executed as a recursive descent through the tree, which creates one new stack frame per node in the tree per thread of execution. Unfortunately, the stack depth then varies depending on runtime variables, and there is the real possibility that a set of sequence data could overrun the stack on the CUDA device, leading to a crash. Fortunately, CUDA stack depth is configurable prior to launching a kernel, so as a solution to this problem, the program dynamically increases the size of the CUDA stack at data initialization.

Once each thread has determined its value of $L^i(G)$, a multiplicative reduction must occur. Recall that the likelihoods at each base pair site are considered to be mutually independent. Just as in the reduction that occurs at the end of the proposal kernel, warp shuffle operators are used to generate one value per warp, which is placed into shared memory and then further reduced by a master thread. One thread per *block* (a higher level of thread organization than a warp) places the value into device memory. If there are enough threads to warrant multiple blocks, the execution thread that calls the data likelihood thread must perform a final reduction of the block-level aggregates. This is also done in serial, but the factor of reduction is so great that it does not add significantly to computation costs. The result of this kernel is the $P(D|G)$ value having been stored for the given genealogy G.

5.2.3 Posterior Likelihood Kernel

The posterior likelihood kernel calculates the value of the relative likelihood of $L(\theta)$ for a given θ and group of genealogy samples G_i, which have been previously generated by Calderhead's method. The form of $L(\theta)$ was taken in Equation 26 to be the mean of the posterior probabilities of the members of G_i, given θ, $P(G_i|\theta)$. Each thread calculates this posterior probability term for a given genealogy in the set of samples. There are as many threads as sampled genealogies, and the thread ID is used as an index on the sample set.

Each individual thread calculates the fraction from Equation 24 for the given genealogy. This is the main computation of the thread. Once this ratio has been computed for each genealogy of the set of samples, a reduction is performed across all threads to find the largest posterior likelihood. This is done to provide a normalizing factor and prevent value overflow, as will be elaborated upon in Section 5.3. Once this normalization is complete, an additive reduction is performed on the normalized posterior values. This is done in the same manner as in the other two kernels.

The result of this kernel is that the expected likelihood of θ as a ratio to the likelihood of the sample-generating θ_0 has been calculated. The kernel as a whole is an implementation of the relative posterior likelihood function which is the function that is subsequently maximized in the MLE calculation in order to find a successor value of θ. This is the input into the 'Maximize' phase of the coalescent genealogy sampler, when it is viewed as an Expectation-Maximization method.

5.3 Underflow Avoidance

Many of the terms being calculated by the kernels — as well as the intermediate terms necessary to compute them — involve very small numbers or fractions with very small numbers in the denominator. The risk on many of these calculations is arithmetic underflow, where the result of a calculation is smaller than the smallest representable floating point. In the case where the extremely small value is a denominator of a fraction, the result can be either a divide by zero fault or an overflow where the value of the fraction becomes a representation of infinity. This can cause bugs that either crash the program or introduce bias into probabilistic calcula-

tions.

This problem is very pronounced on CUDA platforms, which often optimize for speed. The throughput of single-precision operations is much higher in CUDA than that of double-precision operations. However, single-precision implementations of floating point numbers have a much larger minimum value and much smaller maximum value than do double-precision floating point numbers. The trade-off for better floating-point performance is therefore a loss of precision and a higher risk of underflow/overflow.

The program described here avoids this risk by storing any value with a risk of overflow or underflow as a logarithmic value. Instead of storing the value x, for example, the value $log(x)$ is stored in memory. This gives a wider range of computable values, at the loss of some additional precision. Taking M to be the largest possible single-precision floating point, storing logarithms rather than raw values in memory results in values as small as e^{-M} and as large as e^{M} can be stored in memory as results or as intermediate products. If the non-logarithmic value is needed, it can be calculated using the exponential function of the value stored in memory.

For example, suppose that $a = ln(x)$ and $b = ln(y)$ are held in memory. If the value $x \times y$ is needed, that value is calculated logarithmically as $ln(x \times y) = ln(x) + ln(y) = a + b$. If the value $x + y$ is needed, first calculate $k = max(a,b)$, then logarithmically calculate the addition as

$$ln(x+y) = ln\left(e^{a} + e^{b}\right) = ln\left[e^{k}(e^{(a-k)} + e^{(b-k)})\right] = ln\left(e^{(a-k)} + e^{(b-k)}\right) + k. \qquad (32)$$

It is possible that the smaller term in Equation 32 will vanish to zero if it is much smaller than the larger term. This is typically not an issue since it will cause only a minuscule error in terms of the sum of the addition. Furthermore, this method of performing addition will prevent both terms from being reduced to zero in the intermediate calculations, which would result in the sum as a whole being zero.

6 Results

The results of the proof-of-concept implementation of the described method are quantified below. These results comprise both a demonstration of "correctness" in reproducing the original algorithm of LAMARC, as well as a characterization of the performance improvement that results from the parallel implementation.

6.1 Accuracy

The method used to quantify correct operation was to simulate genealogical data from a known θ value, and then compare the results of running θ-estimation using both the production LAMARC package and the proof-of-concept application, mpcgs.

The procedure to produce simulated genealogical data is to first generate a simulated genealogical tree using the program `ms`.[14] This program simulates the evolution of a set of "chromosomes" using a model of genetic drift, as described in Section 2.4. Typically, `ms` provides a set of permutations of different variants of chromosomes to represent different organisms, but it can also include the genealogical tree that was simulated to provide these results. In this case, the output of interest is the tree alone. For example, the command

```
ms 12 1 -T
```

produces a tree in the Newick tree format.[9]. The first argument specifies how many separate samples of genetic information to simulate. This will define the size of the tree. The second argument specifies how many different independent sequences of genetic information `ms` will generate for each sample.

Once a tree has been simulated, the program `seq-gen`[22] can be used to simulate the genealogical sequence data from the tree. This program generates sequences of nucleotides which match the expected genetic separation of sequences that are related by the distances specified in the trees that `ms` generates. The command

```
seq-gen -mF84 -I 200 -s 1.0 < treefile
```

generates a set of genealogical data. The first argument (`-mF84`) specifies the model of mutation that should be used, in this case, the model is F84 (see [9] for details.) The second set of

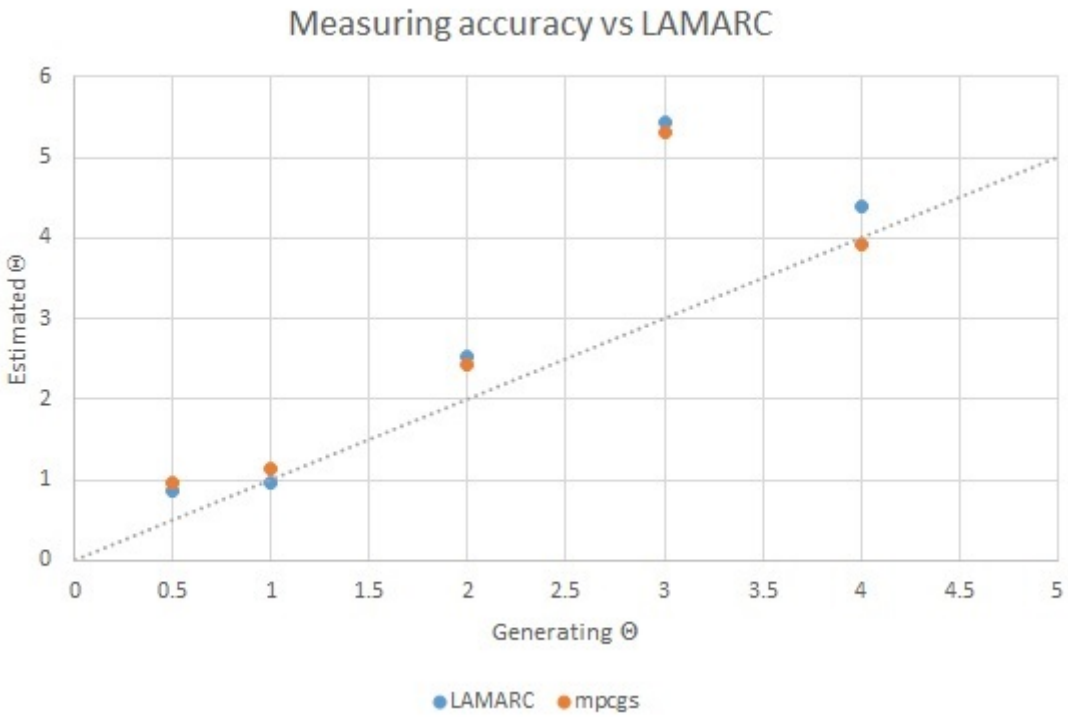

Figure 13: Plotting LAMARC and `mpcgs` accuracy

arguments (`-l 200`) specifies the length of each sequence. The argument given will result in each sequence being 200 nucleotides long. The third argument (`-s 1.0`) specifies that the true θ for simulation purposes is 1.0. The newick tree generated by `ms` is also given as an argument to the `seq-gen` program by input redirection. The `seq-gen` program produces sequence data in the PHYLIP format. LAMARC is packaged with utility programs to convert PHYLIP files into the necessary input format, and `mpcgs` can take these files as direct input.

Table 1: Comparison of LAMARC and `mpcgs` for θ-estimation

True θ	LAMARC	LAMARC StDev	`mpcgs`	`mpcgs` StDev
0.5	0.858	0.024	0.966	0.047
1.0	0.959	0.018	1.131	0.03
2.0	2.521	0.064	2.423	0.136
3.0	5.432	0.064	5.32	0.16
4.0	4.384	0.067	3.913	0.138

See Table 1 for a comparison of the θ estimation of LAMARC and `mpcgs` for different sets of sequence data. This information is also graphed in Figure 13. The comparison should

not be expected to be exact, since these are stochastic processes, and the current version of LAMARC implements a somewhat different proposal mechanism than described in [18]. Additionally, the mutation model assumed by LAMARC and `mpcgs` is subtly different from those provided by `seq-gen`, which accounts for differences (occasionally significant) between the true θ and the value estimated by `mpcgs`. The test of accuracy is the Pearson correlation coefficient, which shows a correlation of $r = 0.905$. This is a level of correlation that can be characterized as very strong[6], especially considering differences in implementation details.

6.2 Performance

The method of testing performance improvement from the application of parallelism (i.e. *speedup*) was to simply compare the runtime of LAMARC and `mpcgs` when provided the same data sets with equivalent scaling parameters. This performance characterization was made across three different dimensions of analysis that could affect performance. The three were the number of genealogical samples generated at each iteration of expectation calculation, the number of sequences in the genealogical data, and the size of the sequences (number of base pairs.)

Table 2: Speedup factor for varying number of samples

# Samples	Speedup
20,000	3.69
30,000	3.8
40,000	3.95
60,000	4.19
80,000	4.27
100,000	4.32

Table 2 and Figure 14 show the speedup relative to the number of genealogical samples taken by each iteration of the expectation calculation phase of the EM-algorithm. In other words, this is the number of samples produced by each run of the coalescent genealogy sampler. Increasing the number of samples increases the length of time the sampler runs, and also

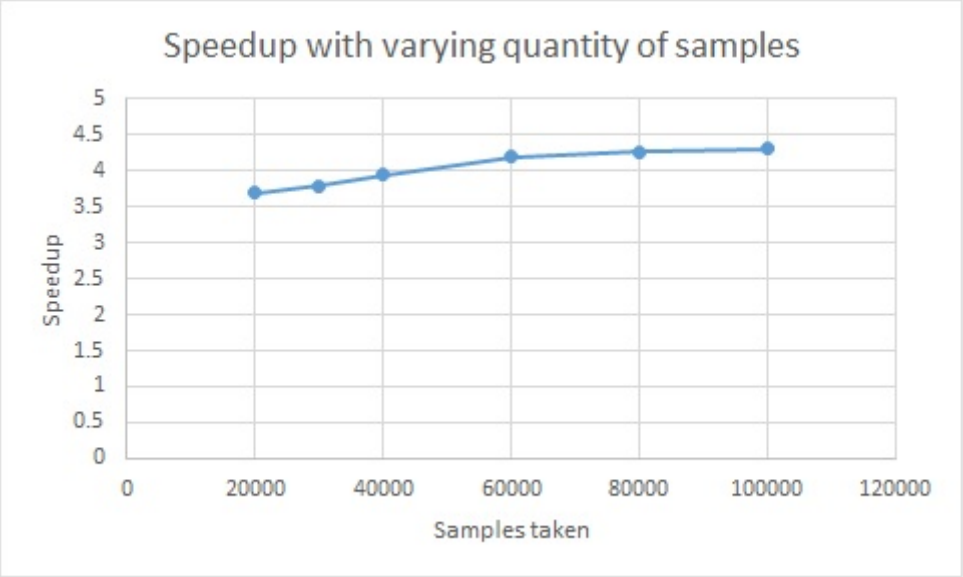

Figure 14: Speedup from varying the number of genealogical tree samples

increases the number of terms in the posterior likelihood function. Speedup does not seem to vary significantly with the number of samples taken.

Table 3: Speedup factor for varying number of sequences

# Sequences	Speedup
12	3.69
24	3.41
36	2.9
48	2.78
60	2.57
84	2.43
108	2.43
132	2.83

Table 3 and Figure 15 show the speedup relative to the number of sequences in the genealogical data being used to produce an estimate of θ. The number of sequences affects the complexity of the genealogies produced by the sampler, since the sequences form the leafs of the genealogical trees. This affects the complexity of the data likelihood calculations, and also the number of intervals that are used to calculate each term of the posterior likelihood. Speedup seems to either not vary or to decline slightly as the number of sequences increases.

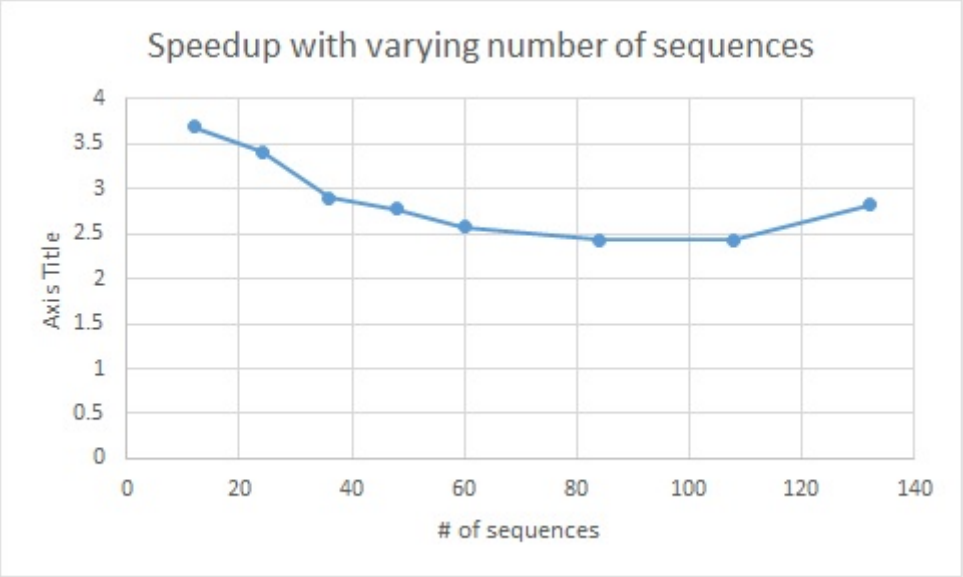

Figure 15: Speedup from varying the number of sequences

Table 4: Speedup factor for varying sequence size

Sequence Size	Speedup
200	3.69
400	5.67
600	7.86
800	10.22
1000	12.63
2000	23.28

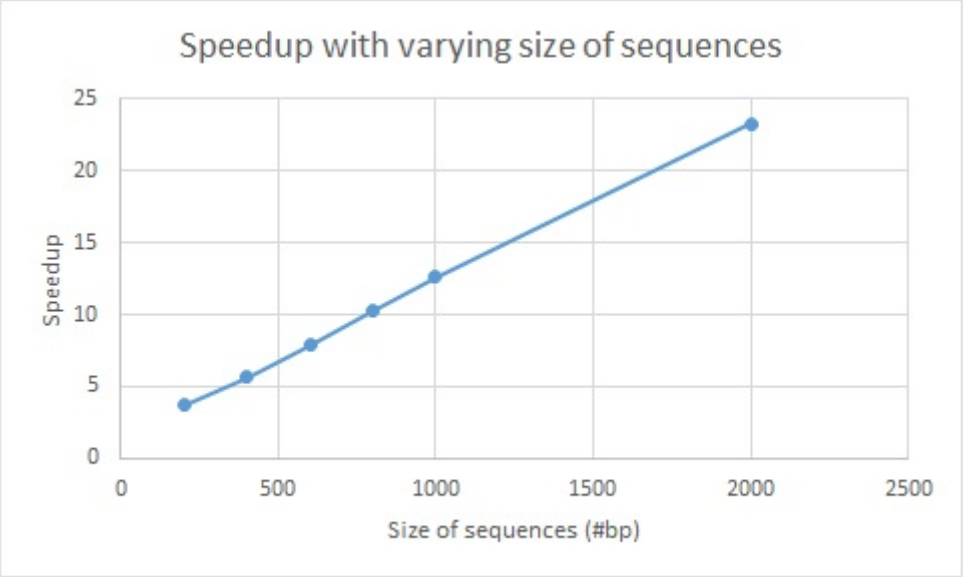

Figure 16: Speedup from varying the sequence size

Table 4 and Figure 16 show the speedup relative to the size of the genealogical sequences. The length of the sequence is the number of base pair positions in each sequence. The length of the sequences affects the amount of computation necessary to calculate the data likelihood for each proposal genealogy in the course of performing Calderhead's method. The speedup seems to increase linearly with sequence length. Limits in implementation scaling prevented testing beyond 2000 base pairs per sequence.

7 Conclusion

The results above show that the program built as a proof of concept for the method described in Section 4 has been successful. A novel method of parallelizing the coalescent genealogy sampler has been found. Moreover, this novel method is free of the inherently sequential burn-in period of previous efforts to apply parallelism to this class of sampler algorithm. As shown in Section 3, this inherently sequential segment would otherwise eventually dominate computation time at high levels of parallelism.

With the burn-in segment parallelized, the idealized computation time for sampling becomes $\frac{B+N}{P}$, where B is the burn-in time, N is the sampling time, and P is the number of processing units on which the program is executing. Of course, in practice, there will be smaller serial components and additional implementation limits on scale, but this new method removes a significant algorithmic limitation. This new method resolves the problem identified in Section 3. Applying this method to large and complex data sets using modern parallel platforms should provide estimates of population parameters much faster than would otherwise be attainable.

These results also show that the method of implementation is most efficient at larger sequence sizes. This is fortunate, since the authors of LAMARC identify increasing the amount of genetic information per sample to be generally the best way to improve study results.[17] The reason for this increased efficiency with sequence size is likely that increasing sequence size primarily increases the number of data likelihood threads executing simultaneously. This more fully utilizes the processing units of the CUDA card, hiding memory latency as there is a larger queue of instructions available for execution.

Future work is necessary to improve the efficiency of this proof of concept implementation. This will require a further redesign of kernel code to reduce data dependencies. As noted earlier, CUDA has very high memory latency relative to instruction throughput, so there are significant performance benefits in replacing any possible memory access with computation. Additional optimization will take the form of tuning various parameters such as the size of the proposal set that Calderhead's method produces and the block size of the data likelihood kernel. Tuning these values will require study across diverse data and execution conditions.

Further work must also be done to enhance the functionality to estimate parameters other than θ. The current LAMARC package is able to estimate multiple parameters for a given set of genealogical samples. Adding a new parameter would require a new proposal kernel to propose genealogies with the posterior probability of that parameter, as well as the ability to calculate that posterior probability for a given genealogy in order to compute the posterior likelihood curve.

Overall, this method is a promising one for future development and scientific use. The application of Calderhead's method to create parallel coalescent genealogy samplers has the potential to greatly reduce the amount of time necessary to perform studies in population genetics by allowing the efficient application of large-scale parallelism.

References

[1] GM Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proc. AFIPS 1967 Spring Joint Computer Conf. 30 (April), Atlantic City, NJ*. 1967, pp. 483–485.

[2] Christophe Andrieu et al. "An Introduction to MCMC for Machine Learning". English. In: *Machine Learning* 50.1-2 (2003), pp. 5–43. ISSN: 0885-6125. DOI: 10.1023/A:1020281327116. URL: http://dx.doi.org/10.1023/A:1020281327116.

[3] Pierre Bremaud. *Markov chains: Gibbs fields, Monte Carlo simulation, and queues*. Vol. 31. Springer Science & Business Media, 1999.

[4] Ben Calderhead. "A general construction for parallelizing Metropolis- Hastings algorithms". In: *Proceedings of the National Academy of Sciences of the United States of America* 111.49 (2014), p. 17408.

[5] Roger Eckhardt. "Stan Ulam, John von Neumann, and the Monte Carlo Method". In: *Los Alamos Science* 15 (1987), pp. 131–136.

[6] James D Evans. *Straightforward statistics for the behavioral sciences*. Brooks/Cole, 1996.

[7] Joseph Felsenstein. "Evolutionary trees from DNA sequences: A maximum likelihood approach". English. In: *Journal of Molecular Evolution* 17.6 (1981), pp. 368–376. ISSN: 0022-2844. DOI: 10.1007/BF01734359. URL: http://dx.doi.org/10.1007/BF01734359.

[8] Joseph Felsenstein. "PHYLIP: phylogenetic inference package, version 3.5 c." In: (1993).

[9] Joseph Felsenstein and Gary A Churchill. "A Hidden Markov Model approach to variation among sites in rate of evolution." In: *Molecular Biology and Evolution* 13.1 (1996), pp. 93–104.

[10] James Allen Fill. "An interruptible algorithm for perfect sampling via Markov chains". In: *The Annals of Applied Probability* 8.1 (1998), pp. 131–162.

[11] J Willard Gibbs. "Elementary Principles in Statistical Mechanics, Developed with especial Reference to the Rational Foundations of Thermodynamics". In: *Bull. Amer. Math. Soc. 12 (1906), 194-210 DOI: http://dx. doi. org/10.1090/S0002-9904-1906-01319-2 PII* (1906), pp. 0002–9904.

[12] Daniel L Hartl, Andrew G Clark, and Andrew G Clark. *Principles of Population Genetics*. Vol. 116. Sinauer associates Sunderland, 1997.

[13] W Keith Hastings. "Monte Carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57.1 (1970), pp. 97–109.

[14] Richard R Hudson. "Generating samples under a Wright–Fisher neutral model of genetic variation". In: *Bioinformatics* 18.2 (2002), pp. 337–338.

[15] JFC Kingman. "The Coalescent". In: *Stocastic Prosess. Appl.* 13 (1982), pp. 235–248.

[16] Mary K Kuhner. "Coalescent genealogy samplers: windows into population history". In: *Trends in Ecology & Evolution* 24.2 (2009), pp. 86–93.

[17] Mary K Kuhner. "LAMARC 2.0: maximum likelihood and Bayesian estimation of population parameters". In: *Bioinformatics* 22.6 (2006), pp. 768–770.

[18] Mary K Kuhner, Jon Yamato, and Joseph Felsenstein. “Estimating effective population size and mutation rate from sequence data using Metropolis-Hastings sampling.” In: *Genetics* 140.4 (1995), pp. 1421–1430.

[19] Makoto Matsumoto and Takuji Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30.

[20] Nicholas Metropolis et al. “Equation of State Calculations by Fast Computing Machines”. In: *J. Chem. Phys* 21 (1953), p. 1087.

[21] NVIDIA. “CUDA C Programming Guide, version 7.5”. In: *NVIDIA Corporation, Santa Clara, CA* (2015).

[22] Andrew Rambaut and Nicholas C Grass. “Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees”. In: *Bioinformatics* 13.3 (1997), pp. 235–238.

[23] Jeffrey S Rosenthal. “Parallel computing and Monte Carlo algorithms”. In: *Far East Journal of Theoretical Statistics* 4 (2000), pp. 207–236.

[24] Mutsuo Saito and Makoto Matsumoto. “Variants of mersenne twister suitable for graphic processors”. In: *ACM Transactions on Mathematical Software (TOMS)* 39.2 (2013), p. 12.

[25] Arun Sethuraman and Jody Hey. “IMa2p–parallel MCMC and inference of ancient demography under the Isolation with migration (IM) model”. In: *Molecular ecology resources* 16.1 (2016), pp. 206–215.